



Tema 4: Corrección y Robustez en Java

Programación Orientada a Objetos

Curso 2009/2010

Begoña Moros Valle



DIS

Departamento de
Informática y Sistemas



Contenido

- Introducción
- Corrección:
 - Aertos
 - Pruebas unitarias
- Robustez:
 - Excepciones
- Diseño por contrato
- Consejos uso de excepciones



Introducción

- **Corrección:**

- Es la capacidad de los productos software de realizar con exactitud su tarea (**cumplir su especificación**).

- **Robustez:**

- Es la capacidad de los productos software de reaccionar adecuadamente ante **situaciones excepcionales**.

- La reutilización y extensibilidad no deben lograrse a expensas de la **fiabilidad** (corrección y robustez).



Especificación

- La **especificación** (formal) de la semántica de una **rutina** está definida por:
 - **Precondiciones**: condiciones para que una rutina funcione correctamente.
 - **Postcondiciones**: describen el efecto de la rutina.
- El estado que deben cumplir los objetos de una clase se denomina **invariante**.
 - Restricciones que deben ser satisfechas por cada objeto tras la ejecución de los métodos y constructores.



Asertos y excepciones

- Java proporciona dos herramientas para tratar con la corrección y robustez del código:
 - **Aserto**: condición que debe cumplirse en el código.
 - **Excepción**: mecanismo para notificar un error durante la ejecución.



Caso de estudio

- Clase **Cuenta**:

- El estado de una cuenta es correcto si tiene un titular (referencia no nula) → **invariante**
- El método **ingreso**() incrementa el saldo de la cuenta → **postcondición**
- El parámetro del método **ingreso**() debe ser siempre positivo → **precondición**
- El método **ingreso**() sólo se puede ejecutar en estado OPERATIVA → **precondición**
- Cada cuenta tiene un código diferente.

- ¿Cómo comprobamos la corrección del código?



Clase Cuenta

```
public class Cuenta {
    private static int ultimoCodigo = 0;
    private int codigo;
    private double saldo;
    private final Persona titular;
    private EstadoCuenta estado;

    public Cuenta(Persona persona, int saldoInicial) {
        codigo = ++ultimoCodigo;
        saldo = saldoInicial;
        titular = persona;
        estado = EstadoCuenta.OPERATIVA;
    }
    public void ingreso(double cantidad) {

        saldo = saldo + cantidad;
    }
    ..
}
```



Aertos

- Construcción proporcionada por el lenguaje que permite **comprobar si una condición se cumple en el código**.
- Declaración de un aserto:
 - **assert** expresión booleana;
 - **assert** expresión booleana: "mensaje de error";
- La **violación de un aserto** (evaluación `false`) provoca un error en la aplicación notificado con una excepción (`AssertionError`).
- Se pueden incluir asertos **en cualquier punto del código** donde queramos asegurar que se cumple una condición → útil para depuración.



Asertos

- Los asertos puede utilizarse para comprobar la corrección del código.
- **Pre y postcondiciones** del método `ingreso()`:

```
/**
 * Ingresa una cantidad en una cuenta operativa
 * @param cantidad valor mayor o igual que cero.
 */
public void ingreso (double cantidad) {

    assert cantidad >= 0: "cantidad negativa";
    assert estado == EstadoCuenta.OPERATIVA: "estado incorrecto";

    saldo = saldo + cantidad;

    assert saldo == old saldo + cantidad: "Fallo postcondición";
}
```



Asertos

■ Postcondición:

- El código anterior no compila, ya que no es posible expresar con un aserto el antiguo valor de un atributo (`old saldo`).
- Para expresar correctamente el aserto deberíamos declarar una **variable local** para almacenar el antiguo valor del atributo.

■ Invariante:

- Podría definirse en un método privado que sea comprobado al final de cada método y constructor.

```
private boolean invariante() {  
    return titular != null;  
}
```



Asertos

```
/**
 * Ingresa una cantidad en una cuenta operativa.
 * @param cantidad valor mayor o igual que cero.
 */
public void ingreso (double cantidad) {

    assert cantidad >= 0: "cantidad negativa";
    assert estado == EstadoCuenta.OPERATIVA: "estado incorrecto";

    double oldSaldo = saldo;

    saldo = saldo + cantidad;

    assert saldo == oldSaldo + cantidad: "Fallo postcondición";
    assert invariante(): "Fallo invariante";
}
```



Asertos. Inconvenientes

- Los asertos en Java son **pobres**.
 - Por ejemplo, no podemos establecer en un aserto el antiguo valor de un atributo (`old saldo`).
 - **No permite** utilizar **cuantificadores** lógicos ni condiciones globales a una clase.
 - Por ejemplo, “cada cuenta debe tener un código único”.
 - Un aserto sólo es **comprobado durante la ejecución** del código.
 - Los asertos **no se heredan**.
 - Por defecto, los asertos están desactivados.
 - Se activan con el parámetro `-ea` de la máquina virtual.
- ➔ **No podemos confiar en los asertos para controlar las precondiciones!!**



Verificación y Pruebas

- **Verificación:** proceso que comprueba que el código sea correcto.
- La verificación puede ser **formal** (lenguajes algebraicos, *Maude*)
- Las **pruebas** del software permiten realizar la **verificación no formal** del código.
- Hay varios tipos de pruebas. Las más relevantes para las clases son las **pruebas unitarias**.
- Java no incluye en el lenguaje soporte para pruebas unitarias.
- Se utiliza una herramienta externa encargada de hacer las pruebas: **JUnit**.



Pruebas Unitarias

- Pruebas encargadas de probar el funcionamiento de un módulo (clase) de la aplicación.
- Una prueba unitaria adopta el **rol del cliente** de la clase y comprueba si los constructores y métodos realizan lo que se espera de ellos (**postcondiciones**).
- La **ejecución** de las pruebas puede ser **automatizada**.
- Es posible definir **baterías de pruebas** (*suite*) que ejecuten todas las pruebas de una aplicación.
- Habitualmente se define una prueba unitaria (clase de pruebas) por cada clase de la aplicación.



JUnit – Asertos

- La herramienta JUnit define métodos para establecer **asertos** sobre el código:
 - **assertEquals()** : comprueba que dos valores sean iguales.
 - **assertNotEquals()** : dos valores son distintos.
 - **assertSame()** : dos objetos son idénticos.
 - **assertTrue()** : validez de una expresión booleana.
 - **assertNotNull()** : referencia no nula.
 - ...

JUnit - Pruebas de la clase Cuenta

- Se crean **métodos de prueba** para los constructores y métodos que cambien el estado del objeto.

```
import junit.framework.TestCase;

public class CuentaTest extends TestCase {

    public void testIngreso() {

        Persona cliente =
            new Persona ("Juan González", "33435332");

        Cuenta cuenta = new Cuenta (cliente, 100);
        cuenta.ingreso(100);

        assertEquals(200, cuenta.getSaldo());
    }
}
```




JUnit – Objetos de prueba

- Para evitar la repetición del código que construye los **objetos de prueba**:
 - Se declaran como **atributos** de la clase de pruebas.
 - El método **setUp()** se encarga de construir los objetos de prueba.
 - El método **setUp()** se ejecuta antes de cada método de prueba.
- ➔ Se asegura que los objetos de prueba siempre estén recién contruidos antes de cada prueba.
- ➔ Los métodos de prueba no interfieren entre sí.

JUnit – Método setUp()

```
public class CuentaTest extends TestCase {  
  
    private Cuenta cuenta;  
  
    @Override  
    protected void setUp() {  
        Persona cliente =  
            new Persona ("Juan González", "33435332");  
  
        cuenta = new Cuenta (cliente, 100);  
    }  
  
    public void testIngreso() {  
        cuenta.ingreso(100);  
  
        assertEquals(200, cuenta.getSaldo());  
    }  
}
```

JUnit – Batería de pruebas

- Se añaden todas las clases de pruebas de la aplicación.

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {

    public static Test suite() {
        TestSuite suite =
            new TestSuite("Test aplicación bancaria");

        suite.addTestSuite(CuentaTest.class);
        suite.addTestSuite(DepositoTest.class);

        return suite;
    }
}
```



Beneficios de las pruebas

- Las pruebas **se ejecutan automáticamente**:
 - No es necesario que una persona interactúe con la interfaz de la aplicación introduciendo valores de prueba.
- Las pruebas **comprueban el comportamiento esperado** de una operación sin importar cómo ha sido implementada.
 - ➔ Los cambios en la implementación que mejoren la calidad del código (*refactorización*) no afectan a las pruebas.
- **Soporte para la extensibilidad**:
 - Tras añadir una nueva funcionalidad a la aplicación las pruebas permiten comprobar que todo el código anterior funciona correctamente.



Excepciones

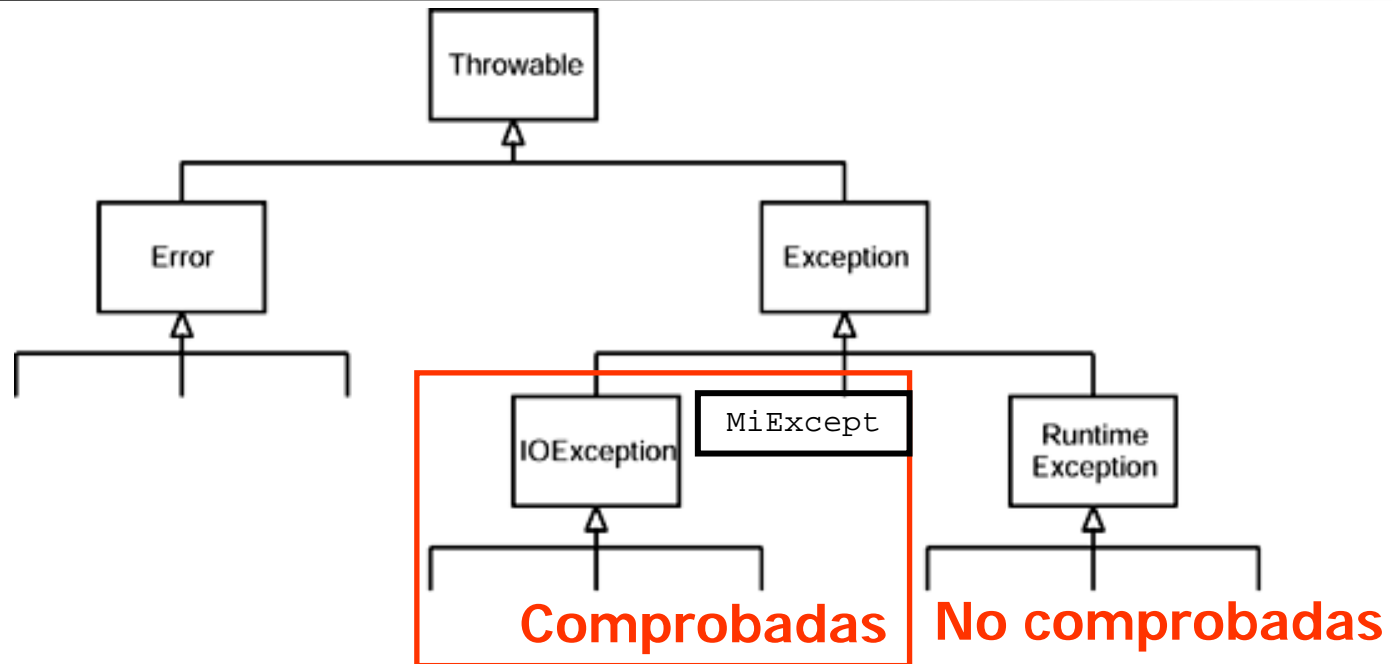
- Mecanismo proporcionado por el lenguaje de programación para **notificar errores en tiempo de ejecución**.
- Soporte para la **robustez** del código.
- La información del error, **excepción**, es un **objeto** que se propaga a todos los objetos afectados por el error.
- Las excepciones pueden tratarse con el propósito de dar una solución al error: **recuperación de errores**.



Situaciones de error

- Habitualmente las excepciones se utilizan en **situaciones de error que no pueden ser resueltas por el programador**:
 - Error en el hardware o sistema operativo: sacar un lápiz de memoria mientras se lee un fichero, la red no está disponible, etc.
 - Fallos en la ejecución de la máquina virtual.
- Además, también se utilizan para la **notificación del uso incorrecto del software**
 - Violación de **precondiciones**
 - Ejemplo: no se puede ingresar en una cuenta una cantidad negativa.

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`.



Tipos de excepciones

- En Java existen dos **tipos de excepciones**:
 - **Comprobadas**:
 - Heredan de la clase **Exception**.
 - Se lanzan en situaciones de error no controladas por el programador. Ejemplo: fallo de red.
 - **No comprobadas** (*runtime*):
 - Heredan de la clase **RuntimeException**.
 - Se lanzan para notificar un uso incorrecto del software (fallo precondiciones).
- El compilador es muy riguroso con el **control de las excepciones comprobadas**
 - Obliga a declararlas en los métodos y a tratarlas en el código de la aplicación.



Excepciones

- En Java las excepciones son objetos y **se definen utilizando una clase**.
- Las excepciones **se declaran** en los métodos y constructores que pueden lanzar esos errores (**throws**).

```
public String leerLinea() throws RedNoDisponible { ... }
```

- Una excepción es **lanzada** utilizando **throw**:

```
throw new RedNoDisponible("La red no está disponible");
```



Definición de una excepción

- Definición de una **excepción comprobada**:

```
public class RedNoDisponible extends Exception {  
    public RedNoDisponible() {  
        super();  
    }  
    public RedNoDisponible(String msg) {  
        super(msg);  
    }  
}
```



Control de precondiciones

- Las **excepciones no comprobadas** pueden utilizarse para el control de precondiciones.
- Ejemplo: método **ingreso** ():
 - No se puede realizar un ingreso en una cuenta si no está operativa y la cantidad es negativa.
- Controlar el cumplimiento de las precondiciones permite la **detección de errores de programación** en el punto en el que se producen:
 - Si no se controlara la cantidad de un ingreso, el estado del objeto cuenta podría quedar corrupto, el balance del banco sería incorrecto, ...



Excepciones no comprobadas

- Una excepción no comprobada se crea definiendo una clase que herede de **RuntimeException**.
- En general, no es necesario crear nuevas excepciones no comprobadas, ya que el lenguaje proporciona varias:
 - **NullPointerException**: excepción de uso de una referencia nula.
 - **IllegalArgumentException**: se está estableciendo un argumento incorrecto a un método.
 - **IllegalStateException**: la aplicación de un método no es permitida por el estado del objeto.
 - ...



Control de precondiciones

- Habitualmente se comprueban dos **tipos de precondiciones**:

- **Parámetros**: los parámetros de un método son los correctos.

Ejemplo: cantidad positiva en método **ingreso**().

- **Estado**: un método no puede ser invocado en el estado del objeto.

Ejemplo: el método **ingreso**() sólo puede ser ejecutado si la cuenta está operativa.



Control de precondiciones

```
/**
 * Ingresa una cantidad en una cuenta operativa.
 * @param cantidad valor mayor o igual que cero.
 */
public void ingreso (double cantidad) {

    if (cantidad < 0)
        throw new IllegalArgumentException("Cantidad negativa");

    if (estado != EstadoCuenta.OPERATIVA)
        throw new IllegalStateException("Estado incorrecto");

    saldo = saldo + cantidad;

    // Las postcondiciones e invariante se comprueban
    // con pruebas JUnit
}
```



Control de precondiciones

- El incumplimiento de una precondición es entendido como un error de programación y no como una situación anómala de la ejecución.
- **No es obligatorio que un método declare las excepciones *runtime*.**
- **No es obligatorio dar tratamiento** a esas excepciones, ya que se supone que “no deberían” ocurrir.



Caso de estudio

- **Navegador web** que visualiza páginas almacenadas en servidores web de internet.
- El navegador web define el método **visualiza ()** encargado de **representar una página**.
- El método **visualiza ()** hace uso de la clase **Conexion** encargada de establecer una conexión con un servidor web y recuperar un recurso.



Caso de estudio

- La clase **Conexion**:
 - Establece una conexión con el servidor web y abre el recurso cuando se construye el objeto.
 - Ofrece un método, **leerLinea()**, que devuelve las líneas del fichero.
 - Define un método para cerrar la conexión.
- El programador de la clase **Conexion** se enfrenta a las siguientes **situaciones de error**:
 - *La red no está activada.* Esta situación afecta al constructor y al método que lee las líneas.
 - *No se puede resolver el nombre del servidor.* Afecta al constructor.
 - *No existe el recurso en el servidor.* Afecta al constructor.



Caso de estudio

- Para cada una de esas situaciones de error se definen **excepciones**:
 - **RedNoDisponible**, **ServidorNoEncontrado**, **RecursoNoExiste**.
- Declara las excepciones en la clase:

```
public class Conexion {  
  
    public Conexion(String url)  
        throws RedNoDisponible, ServidorNoEncontrado,  
            RecursoNoDisponible { ... }  
  
    public String leerLinea() throws RedNoDisponible { ... }  
  
    public void cerrar() { ... }  
}
```



Caso de estudio

- En el navegador, el método **visualiza** () realiza los siguientes pasos:
 - Crea un objeto conexión.
 - Lee las líneas del fichero para construir la representación de la página.
 - Representa la página.
 - Cierra la conexión.



Caso de estudio

```
public void visualiza(String url) {  
  
    Conexion conexion = new Conexion(url);  
  
    String linea;  
    do {  
        linea = conexion.leerLinea();  
        if (linea != null) {  
            construyeRepresentacion(linea);  
        }  
    } while (linea != null);  
  
    representacion();  
    conexion.cerrar();  
}
```



Caso de estudio

- ¿Cuál es el **tratamiento de las excepciones** de la clase **Conexion**?
- Al crear la conexión:
 - *Red no disponible*: realizar varios reintentos esperando un intervalo de tiempo entre ellos. ¿Y si no se recupera la red?
 - *¿Servidor no encontrado?*
 - *¿Recurso no disponible?*



Tratamiento de excepciones

- Java ofrece la construcción **try-catch** para tratar las excepciones que puedan producirse en el código.
- Esta construcción está formada por:
 - **Bloque try**: bloque que encierra código que puede lanzar excepciones.
 - **Bloques catch** o **manejadores**: uno o varios bloques encargados de dar tratamiento a las excepciones.
 - **Bloque finally**: bloque que siempre se ejecuta, se produzca o no excepción.
- En Java las excepciones son objetos.
- Al producirse un error en el bloque `try` se revisan por orden de declaración los manejadores que pueden tratar el error → **Sólo un manejador trata el error.**
- Esta comprobación utiliza la **compatibilidad de tipos.**

Tratamiento de excepciones

```
Conexion conexión = null;
int intentos = 0;

while (intentos < 20) {
    try {
        conexion = new Conexion(url);
        break;
    } catch (RedNoDisponible e) {
        Thread.sleep(1000); // Espera un segundo
        intentos++;
    } catch (ServidorNoEncontrado e) {
        // ¿Qué hacemos aquí?
    } catch (RecursoNoDisponible e) {
        // ¿Qué hacemos aquí?
    }
}
if (intentos == 20) {
    // ¿Qué hacemos aquí?
}
```



Tratamiento de excepciones

- Para las siguientes excepciones no se encuentra tratamiento de error:
 - Se alcanza el número máximo de intentos.
 - El servidor no existe.
 - El recurso no existe en el servidor.
- El método **visualiza** () del **Navegador** debe:
 - Declarar que puede lanzar las tres excepciones.
 - Para los casos *servidor* y *recurso no encontrado*, sencillamente no declara bloques `catch`
 - Las excepciones escaparían del método.
 - Al alcanzar el número máximo de reintentos lanza la excepción `RedNoDisponible` (fallo postcondición).



Tratamiento de excepciones

```
public void visualiza(String url)
    throws RedNoDisponible, ServidorNoEncontrado,
           RecursoNoDisponible {

    Conexion conexion = null;
    int intentos = 0;
    while (intentos < 20) {
        try {
            conexion = new Conexion(url);
            break;
        } catch (RedNoDisponible e) {
            Thread.sleep(1000); // Espera un segundo
            intentos++;
        }
    }
    if (intentos == 20) {
        throw new RedNoDisponible("La red no está disponible");
    }
}
```

Relanzar una excepción

- Es posible volver a lanzar una excepción utilizando **throw** → es tratada y sale del bloque try-catch.
- Ejemplo: dentro del bucle si se alcanzan el máximo de reintentos se relanza la excepción.

```
while (intentos < 20) {  
    try {  
        conexion = new Conexion(url);  
        break;  
    } catch (RedNoDisponible e) {  
        Thread.sleep(1000); // Espera un segundo  
        intentos++;  
        if (intentos == 20) {  
            throw e;  
        }  
    }  
}
```



Excepciones no tratadas

- Una **excepción no tratada** aborta la ejecución de un método en el punto en que se produce.
- Asimismo, el **lanzamiento de una excepción** también finaliza la ejecución del método en el punto en el que se lanza.
- Ejemplo:
 - La excepción **ServidorNoEncontrado** se produce en el constructor de la clase **Conexion**.
 - La excepción llega al método **visualiza()** que no da un tratamiento a esa excepción en el bloque `try-catch`.
 - La excepción escapa al método **visualiza()** y **se propaga al método que lo ha llamado**.



Propagación de excepciones

- Es posible que una excepción pueda propagarse a través de varios métodos.
- **Si una excepción escapa al método `main()` de la aplicación, el programa finaliza con un error.**
- Las excepciones para las que no existe recuperación de error en el código suelen propagarse hasta la **interfaz con el usuario** (pantalla, página web).
- En la interfaz se notifica al usuario el error para que lo resuelva:
 - Ejemplo: el servidor no existe. El usuario comprueba si lo ha escrito mal para volver a intentar visualizar la página.



Excepciones significativas

- Un tipo de tratamiento de excepciones es lanzar una **excepción más significativa**.
- Es útil para ocultar errores de bajo nivel:
 - No se puede abrir el socket de red, error de entrada/salida, etc.
- Ejemplo:
 - El método `visualiza()` podría definir la excepción `ErrorVisualizacion` representando cualquier tipo de error producido en el método.
 - En lugar de dejar escapar las excepciones, se atrapan y se lanzaría la nueva excepción.
 - Para el ejemplo no resultaría útil, ya que las tres excepciones lanzadas por `Conexion` son significativas del tipo de error que se está produciendo.



Excepciones comprobadas

- El compilador es muy riguroso con el **control de las excepciones comprobadas**.
- Si se utiliza un método que puede lanzar una excepción comprobada, el compilador permite sólo dos opciones:
 - Dar tratamiento al error en un bloque `try-catch`.
 - Declarar que el método puede producir ese error (`throws`).



Uso de excepciones

- Las **excepciones comprobadas** son utilizadas para notificar que un método no puede cumplir con su responsabilidad por razones ajenas al programador (**postcondición**).
- Las **excepciones no comprobadas** se utilizan para indicar que una clase se está utilizando incorrectamente (errores de programación, violación de **precondiciones**).



Valores de retorno

- Habitualmente, para notificar que un método no puede cumplir su tarea se han utilizado valores de retorno.
 - Ejemplo: boolean **visualiza**(String url)
 - Si se produce un error, se notifica devolviendo un valor `false`.
- En el caso de que el método deba devolver un valor, se utiliza un valor especial:
 - Ejemplo: String **leerLinea**()
 - Si no hay líneas disponibles devuelve un valor `null`.



Valores de retorno

- La estrategia del valor de retorno está limitada y tiene **problemas**:
 - ¿Qué valor de retorno tiene un constructor?
 - A veces no se puede devolver un valor especial.
 - Ejemplo: `int parseInt(String valor)`
 - Devolver un valor booleano es poco significativo.
 - Ejemplo, el método `visualiza()` produce tres tipos de errores.
 - Java permite ignorar el valor de retorno al llamar a un método.
- **¿Excepciones o valores de retorno?**

Se aplica el sentido común. Depende del nivel de gravedad del error y la necesidad de información.



Diseño por contrato

- Técnica de programación que define claramente las responsabilidades de la clase que ofrece un servicio y el cliente que hace uso de él.
 - **Responsabilidad del Cliente:**
 - Debe llamar a los métodos y constructores de la clase “asegurando” que se cumplen las **precondiciones**.
 - **Responsabilidad del Servidor (clase):**
 - Asegura que las operaciones cumplen con su especificación, es decir, que se cumplen sus **postcondiciones**.
- Si el cliente no cumple con su responsabilidad, el servidor no realiza su tarea y lanza una excepción (runtime)
 - Si el cliente cumple con su responsabilidad y el servidor no puede realizar su tarea debe lanzar una excepción para notificarlo (comprobada)



Diseño por contrato

- **Asegurar que se cumple la precondition** antes de llamar a un método no significa tener que comprobarla:
 - El contexto del código puede indicar que se cumple la precondition.
 - Ejemplo: si se realiza un ingreso en una cuenta justo de después de crearla, podemos asegurar que se cumple la precondition de estado.

- **El cliente debe poder comprobar las precondiciones**:
 - Ejemplo: la clase `Cuenta` debe ofrecer un método para consultar su estado.



Excepciones y Herencia

- Al redefinir un método heredado **podemos modificar la declaración de las excepciones comprobadas** lanzadas (`throws`).
- Sólo es posible **reducir** la lista de excepciones.
- No se puede incluir una excepción comprobada que no lance el método de la clase padre.
- Es posible indicar una excepción más específica que la que se hereda:
 - Ejemplo: en la clase padre el método lanza `IOException` y la redefinición `FileNotFoundException` que es un subtipo.
- Las restricciones sólo afectan a las excepciones comprobadas.



Consejos

- **Documentar las precondiciones** de los métodos.
- **Controlar las precondiciones** al comienzo del método y lanzar una **excepción *runtime*** significativa si no se cumplen.
- Aplica **Diseño por contrato**: el cliente debe asegurar que se cumplen las precondiciones.
- Define y usa **excepciones comprobadas** en los métodos que no puedan cumplir su tarea (postcondición) → razones ajenas al programador
- **No debemos silenciar el tratamiento de una excepción** (manejador de excepción vacío)
→ Antes es preferible no tratarla.



Consejos

- Si el tratamiento de error es notificar al usuario, la **notificación depende de la interfaz** (textual, gráfica).
- Al entregar una aplicación, asegura que la ejecución no finalice por causa de una excepción.
 - Declara el código del método `main()` dentro de un bloque `try-catch`. El tratamiento de los errores inesperados debe ser notificado al usuario.
- Al lanzar una excepción **establece el mensaje de error**. El mensaje de error puede ser mostrado al usuario:
 - `e.getMessage()` ;



Consejos

- **No utilizar excepciones como mecanismo para controlar la ejecución del código.**
 - Una excepción no debe sustituir a una comprobación.
- Si varias instrucciones lanzan excepciones con el mismo tratamiento, es recomendable que un solo bloque `try-catch` envuelva a todas ellas.
- Para depurar una excepción muestra la **traza de del error**:
 - `e.printStackTrace()`