Tema 3. Tratamiento de Excepciones

Vicente Benjumea García

Programación Orientada a Objetos
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Tema 3. Tratamiento de excepciones

- Software tolerante a fallos. El concepto de excepción
- Lanzamiento de excepciones
- Propagación de excepciones
- Captura y tratamiento de excepciones
- Excepciones predefinidas
- Definición de nuevas excepciones

Gestión de errores

- Durante la ejecución de un programa se pueden producir **errores** con diversos **niveles** de severidad:
  - Un índice fuera de rango.
  - Una división por cero.
  - Un fichero que no existe.
  - Una conexión de red que falla.
  - etc.
- El código sería **poco legible** si continuamente tuviésemos que comprobar las posibles **condiciones de error** sentencia tras sentencia.
- Consideremos el siguiente (pseudo)código del método que lee un fichero y copia su contenido en memoria.

```
void leerFichero() {
    abrir el fichero;                   // ¿Qué pasa si el fichero no puede abrirse?
    determinar la longitud del fichero; // ¿Qué pasa si no es posible?
    reservar la memoria suficiente;     // ¿Qué pasa si no hay memoria suficiente?
    copiar el fichero en memoria;       // ¿Qué pasa si falla la lectura?
    cerrar el fichero;                  // ¿Qué pasa si el fichero no puede cerrarse?
}
```

Gestión de errores

- El control de errores hace que el código sea difícil de leer y modificar. **Se pierde el flujo lógico de ejecución**.

```
TipoDeCodigoDeError leerFichero() {
    TipoDeCodigoDeError codigoDeError = 0;
    // ABRIR EL FICHERO
    if (el fichero está abierto) {
        // DETERMINAR LA LONGITUD DEL FICHERO
        if (se consigue la longitud del fichero) {
            // RESERVAR LA MEMORIA SUFICIENTE
            if (se consigue la memoria) {
                // COPIAR EL FICHERO EN MEMORIA
                if (falla la lectura) {
                    codigoDeError = -1;
                }
            } else {
                codigoDeError = -2;
            }
        } else {
            codigoDeError = -3;
        }
        // CERRAR EL FICHERO
        if (codigoDeError == 0 && el fichero no se cerró) {
            codigoDeError = -4;
        }
    } else {
        codigoDeError = -5;
    }
    return codigoDeError;
}
```

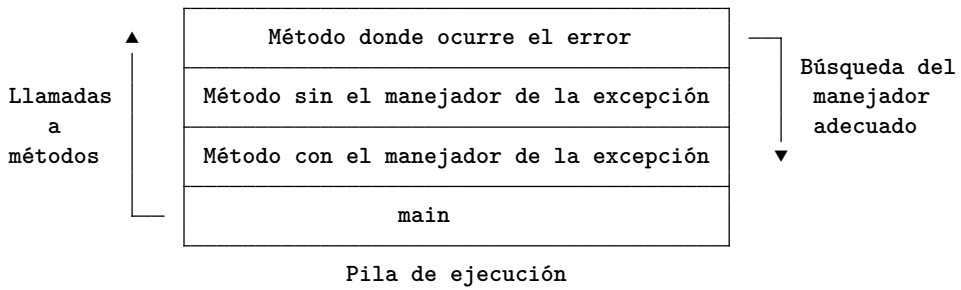
El concepto de "excepción"

- El mecanismo de las **excepciones** proporciona una forma clara de **gestionar** posibles **errores** sin oscurecer el código.
- Las excepciones no nos liberan de hacer la detección, de informar y de manejar los errores, pero nos permiten escribir el **flujo principal** de nuestro código en un sitio y de tratar los casos **excepcionales separadamente**.

```
void leerFichero() {
    try {
        abrir el fichero;
        determinar la longitud del fichero;
        reservar la memoria suficiente;
        copiar el fichero en memoria;
        cerrar el fichero;
    } catch (falló la apertura del fichero) {
        // ...
    } catch (falló el cálculo de la longitud del fichero) {
        // ...
    } catch (falló la reserva de memoria) {
        // ...
    } catch (falló la lectura del fichero) {
        // ...
    } catch (falló el cierre del fichero) {
        // ...
    }
}
```

El concepto de "excepción"

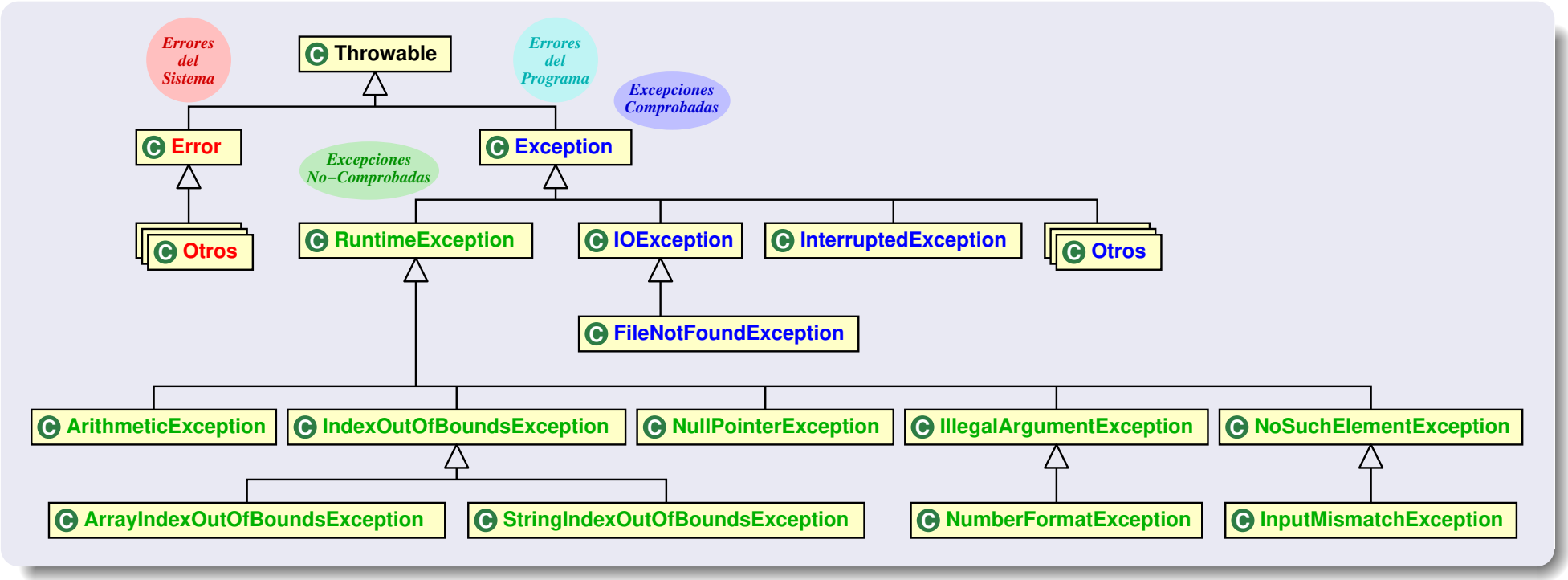
- Una **excepción** es un evento (una señal) que **interrumpe** el flujo normal de instrucciones durante la **ejecución** de un programa como consecuencia de un **error** (condición excepcional).
- Cuando ocurre un error en un método, entonces se **crea un objeto** excepción y lo **lanza** (`throw`) al sistema de ejecución.
  - Este objeto contiene **información sobre el error**, incluido su tipo y el estado del programa donde ocurrió.
- El sistema de ejecución recorre la **pila de llamadas buscando** un método que contenga un bloque de código adecuado que maneje la excepción (**manejador de excepción**).
- Si el sistema **no encuentra** un manejador adecuado, entonces el programa **termina** (mostrando información detallada sobre ella).



El concepto de "excepción"

- Hay **tres puntos de vista** para las excepciones:
  1. El que **lanza** (eleva) la excepción:
     - Durante la ejecución de un método, se encuentra una **situación anómala** que **no sabe cómo resolver** e impide que el método **cumpla** su tarea, por lo tanto **crea** una excepción y la **lanza**, para informar al *nivel superior* de la situación anómala y de que la tarea no pudo completarse.
  2. El que **propaga** la excepción:
     - Durante la ejecución de un método, **recibe** una excepción, **no sabe como tratarla** y recuperar el error, por lo que el método no puede completar su tarea, y por lo tanto, **propaga la excepción**, para informar al *nivel superior* de la situación anómala y de que la tarea no pudo completarse.
  3. El que **maneja** (captura) la excepción.
     - Durante la ejecución de un método, **recibe** una excepción, y **sí sabe cómo resolver** la situación que provoca un **determinado tipo** de excepción, **dispone de un tratamiento** para ella y recupera el error para poder completar su tarea.
- Un mismo método puede actuar desde los tres puntos de vista
  - Captura unas excepciones, propaga y lanza otras.

La clase Throwable y sus subclases



Algunas excepciones lanzadas por el sistema

```
int x = 5 / 0 ;                 // ArithmeticException("/ by zero");

String f = null;
char g = f.charAt(9);           // NullPointerException();

int[] array = new int[5];
int b = array[9];               // ArrayIndexOutOfBoundsException("Index 9 out of bounds for leng

String c = "Hola";
char d = c.charAt(9);           // StringIndexOutOfBoundsException("String index out of range: 9"

int h = Integer.parseInt("123.45");     // NumberFormatException("For input string: \"123.45\

double i = Double.parseDouble("hola");  // NumberFormatException("For input string: \"hola\""

Scanner scanner = new Scanner("");
String k = scanner.next();              // NoSuchElementException();

Scanner scanner = new Scanner("hola");
int j = scanner.nextInt();              // InputMismatchException();

Scanner scanner = new Scanner(new File("datos.txt"));
// FileNotFoundException("datos.txt (No existe el fichero o el directorio)");
```

La clase Throwable

- Sólo los objetos que son **instancias** de la clase `Throwable` (o subclases) pueden ser **lanzados** (`throw`), y sólo éstos pueden ser manejados (`catch`).
- Por convenio, la clase `Throwable` y sus subclases tienen dos **constructores**: uno sin argumentos y otro con un argumento de tipo `String`, el cual puede ser usado para producir mensajes de error.
- Un objeto de la clase `Throwable` contiene el **estado de la pila de ejecución** (de su *hebra*) en el momento en que fue creado.
- La clase `Throwable` proporciona los siguientes métodos públicos:
  - `String getMessage()`: devuelve el texto con el mensaje de error del objeto.
  - `String toString()`: devuelve una descripción textual del objeto.
  - `void printStackTrace()`: imprime el objeto y la traza de ejecución en `System.err`.

```
public class Throwable {
    public Throwable() ;
    public Throwable(String msg) ;
    public String getMessage() ;
    public String toString() ;
    public void printStackTrace() ;
}
```

Lanzar una excepción: la sentencia `throw`

- Una excepción es una **instancia** de una clase, que se crea con el operador `new`.
  - La sentencia `throw` permite **lanzar** un *objeto Excepción*.
  - **Interrumpe** el flujo de ejecución normal y se procede a la **búsqueda** de un manejador para esa excepción, es decir, *"alguien que la trate"*. Por ejemplo:

```
public class Datos {
    private int[] datos;
    public Datos() {
        datos = new int[(int)(10 * Math.random())]; // puede ser cero
        for (int i = 0; i < datos.length; ++i) {
            datos[i] = (int)(100 * Math.random());  // [0 ... 99]
        }
    }
    public double media() {          // LANZA LA EXCEPCIÓN
        if (datos.length == 0) {
            throw new RuntimeException("coleccion vacia");
        }
        double suma = 0;
        for (int i = 0; i < datos.length; ++i) {
            suma += datos[i];
        }
        return suma / datos.length;
    }
}
```

Propagar una excepción

- Si durante la ejecución de un determinado método se lanza (o recibe) una excepción, si la excepción **no se captura**, entonces la excepción será **automáticamente propagada** al nivel superior.

```
public class Datos {
    // ...
    public double desvTipica() {      // PROPAGA LA EXCEPCIÓN
        double m = media();           // puede lanzar una excepción
        double suma = 0;
        for (int i = 0; i < datos.length; ++i) {
            suma += Math.pow(datos[i] - m, 2);
        }
        return Math.sqrt(suma / datos.length);
    }
}
```

Captura de excepciones. La sentencia `try-catch-finally` (I)

La sentencia `try-catch-finally`

- Las excepciones que se lancen durante la ejecución del código del bloque `try` serán capturadas en el bloque `catch` correspondiente, o serán propagadas.
- El bloque `catch` define código que se ejecuta si dentro del bloque `try` correspondiente **se ha lanzado una excepción** especificada en su lista de excepciones capturadas, o de cualquier clase derivada de ellas.
- El bloque `finally` define código que **se ejecuta siempre** que se ejecuta el bloque `try` correspondiente.

- El bloque `try` puede especificar la creación de algunos **recursos** (`PrintWriter`, `BufferedReader`, `Scanner`, etc) que serán **cerrados automáticamente** al finalizar el bloque `try` (antes del `catch`/`finally`).

- Un bloque `try` puede tener cero o múltiples bloques `catch`, pero sólo uno o ningún bloque `finally`.

- Los bloques `catch` se **comprueban según el orden** especificado en el programa, según el tipo especificado y el tipo del objeto excepción a capturar.

Captura de excepciones. La sentencia `try-catch-finally` (II)

Capturar excepciones

```
try {
    // ... Las excepciones que se lancen podrían ser capturadas si el tipo es adecuado
} catch (RuntimeException | IOException e) {
    // ... Captura y tratamiento de excepciones RuntimeException, IOException y derivadas
} catch (Exception e) {
    // ... Captura y tratamiento de excepciones Exception y derivadas
} finally {
    // ... Código que se ejecuta siempre que se ejecuta el bloque try
}
```

Capturar excepciones y cerrar recursos automáticamente

```
try (PrintWriter pw = new PrintWriter(nombreFichero)) {
    // ... Las excepciones que se lancen podrían ser capturadas si el tipo es adecuado
} catch (Exception e) {
    // ... Captura y tratamiento de excepciones Exception y derivadas
} finally {
    // ... Código que se ejecuta siempre que se ejecuta el bloque try
}
// El recurso pw (PrintWriter) se cierra automáticamente
```

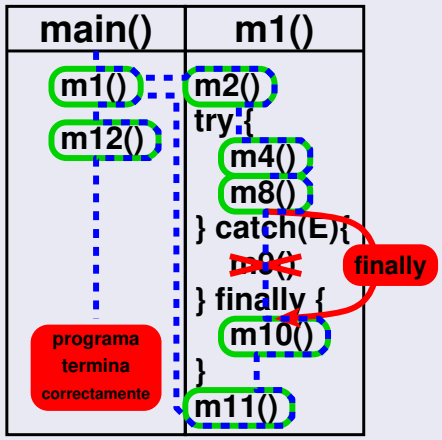
Captura de excepciones relacionadas jerárquicamente

- Se pueden establecer manejadores para **jerarquías** de excepciones.
  - El bloque `catch` captura cualquier objeto excepción de la clase especificada en la cláusula, o de cualquier **subclase** de ella. Polimorfismo en la captura de excepciones.
  - Los bloques `catch` se comprueban según el orden especificado en el programa, y atendiendo al tipo dinámico del objeto excepción a capturar y al tipo especificado.

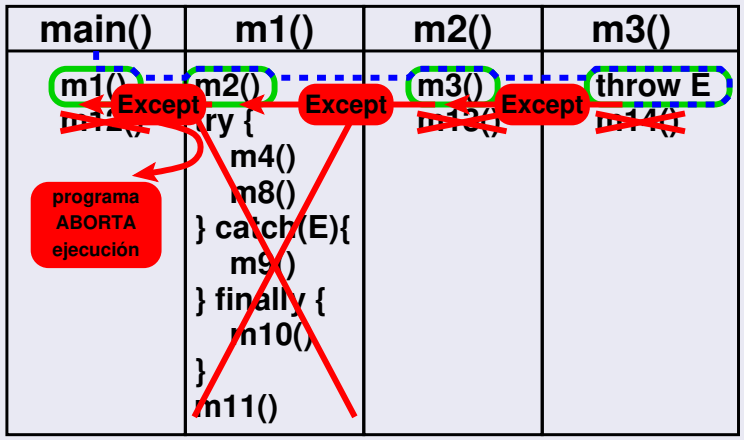
```
public class Test {
    public static void main(String[] args) {
        try {
            int bb = Integer.parseInt(args[0]);
            int bn = Integer.parseInt(args[1]);
            // ...
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Uso: Test <Int> <Int>");
        } catch (NumberFormatException e) {
            System.out.println("Los argumentos deben ser números enteros");
        } catch (RuntimeException e) {
            // Captura todas las excepciones subclases de RuntimeException
            System.out.println("Alguna excepción distinta");
        } finally {
            System.out.println("El programa sigue aquí");
        }
    }
}
```

El flujo de ejecución en presencia de excepciones

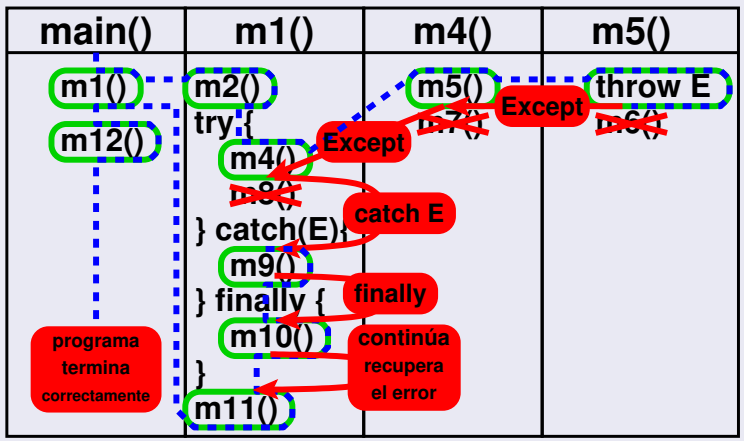
Escenario 1: Ejecución correcta



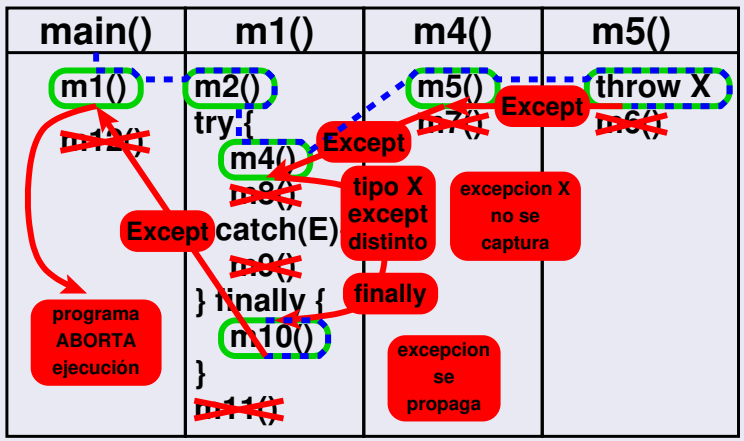
Escenario 2: Excepción E no capturada



Escenario 3: Excepción E capturada



Escenario 4: Excepción X no capturada



Captura de excepciones. Ejemplo (I)

- Calcular la media y desviación típica de una colección aleatoria de números.

```
import java.util.Arrays;
public class Datos {
    private int[] datos;
    public Datos() {
        datos = new int[(int)(10 * Math.random())]; // puede ser cero
        for (int i = 0; i < datos.length; ++i) {
            datos[i] = (int)(100 * Math.random());
        }
    }
    public double media() { // LANZA LA EXCEPCIÓN
        if (datos.length == 0) { throw new RuntimeException("coleccion vacia"); }
        double suma = 0;
        for (int i = 0; i < datos.length; ++i) {
            suma += datos[i];
        }
        return suma / datos.length;
    }
    public double desvTipica() { // PROPAGA LA EXCEPCIÓN
        double m = media();
        double suma = 0;
        for (int i = 0; i < datos.length; ++i) {
            suma += Math.pow(datos[i] - m, 2);
        }
        return Math.sqrt(suma / datos.length);
    }
    public String toString() { // CAPTURA LA EXCEPCIÓN
        String str;
        try {
            str = "{ " + Arrays.toString(datos) + " : " + media() + " : " + desvTipica() + " }";
        } catch (RuntimeException e) {
            str = "{ " + Arrays.toString(datos) + " }";
        }
        return str;
    }
}
```

Captura de excepciones. Ejemplo (II)

- Los manejadores pueden hacer más que simplemente mostrar información en consola, pueden **recuperar** la situación de error.
- Generar una colección aleatoria de números. En caso de datos incorrectos (colección vacía o desviación típica mayor que 1) volver a generar la colección.

```
public class Main {
    public static void main(String[] args) {
        boolean ok = false;
        Datos d = null;
        do {
            try {                                  // VIGILA LAS EXCEPCIONES
                System.out.println("Generando datos ...");
                d = new Datos();
                double dt = d.desvTipica(); // Puede lanzar RuntimeException
                ok = (0 <= dt && dt <= 1);
            } catch (RuntimeException e) {  // CAPTURA LA EXCEPCIÓN
                ok = false;
            }
        } while (! ok);
        System.out.println("Datos: "+d);     // datos, media y desviación típica
    }
}
```

El bloque `finally` (I)

- El bloque `finally` **se ejecuta siempre** que se ejecuta el bloque `try` correspondiente. Su cometido es la **recuperación de recursos** y **restablecer un estado correcto** en la ejecución.

```
public class CajaFuerte {
    private static final int CLAVE = 11; // 7
    private boolean puertaAbierta;
    private double saldo;
    public CajaFuerte() {
        puertaAbierta = false; saldo = 0;
    }
    public void abrirPuerta(int c) {
        if (CLAVE != Math.pow(c, 5) % 221) {
            throw new RuntimeException("Clave errónea");
        }
        puertaAbierta = true;
    }
    public void operacion(double c) {
        if ( ! puertaAbierta ) {
            throw new RuntimeException("Puerta cerrada");
        }
        if (saldo+c < 0) {
            throw new RuntimeException("Saldo insuficiente");
        }
        saldo += c;
    }
    public String toString() {
        return "Caja[" + (puertaAbierta ? "A, " : "C, ") + saldo +"]";
    }
}
```

```
public void cerrarPuerta() {
    puertaAbierta = false;
}
```

El bloque `finally` (II)

- El bloque **`finally`** **se ejecuta siempre** que se ejecuta el bloque **`try`** correspondiente. Su cometido es la **recuperación de recursos** y **restablecer un estado correcto** en la ejecución.

```
public class Main1 {
    private static void operaciones(CajaFuerte caja, int clave, int[] ops) {
        caja.abrirPuerta(clave);
        for (int c : ops) {                                  // Caja[A, 1000.0]
            caja.operacion(c);                               // Caja[A, 900.0]
            System.out.println(caja);                        // Caja[A, 700.0]
        }
        caja.cerrarPuerta(); // Si se lanza una excepcion, entonces no se cerrará
    }
    public static void main(String[] args) {
        int[] ops = { 1000, -100, -200, -2000, 100 };
        CajaFuerte caja = new CajaFuerte();
        try {
            operaciones(caja, Integer.parseInt(args[0]), ops); // args[0] = "7"
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Uso: Main <Int>");
        } catch (NumberFormatException e) {
            System.out.println("El argumento debe ser entero");
        } catch (Exception e) {
            e.printStackTrace();                             // RTE: Saldo insuficiente
        } finally {
            System.out.println("Estado final: "+caja);       // Estado final: Caja[A, 700.0]
        }                                                    // La Caja queda ABIERTA
    }
}
```

El bloque `finally` (III)

- El bloque **`finally`** **se ejecuta siempre** que se ejecuta el bloque **`try`** correspondiente. Su cometido es la **recuperación de recursos** y **restablecer un estado correcto** en la ejecución.

```
public class Main2 {
    private static void operaciones(CajaFuerte caja, int clave, int[] ops) {
        caja.abrirPuerta(clave);
        try {
            for (int c : ops) {                                  // Caja[A, 1000.0]
                caja.operacion(c);                               // Caja[A, 900.0]
                System.out.println(caja);                        // Caja[A, 700.0]
            }
        } finally {
            caja.cerrarPuerta();            // GARANTIZA QUE LA PUERTA SE CIERRA
        }
    }
    public static void main(String[] args) {
        int[] ops = { 1000, -100, -200, -2000, 100 };
        CajaFuerte caja = new CajaFuerte();
        try {
            operaciones(caja, Integer.parseInt(args[0]), ops); // args[0] = "7"
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Uso: Main <Int>");
        } catch (NumberFormatException e) {
            System.out.println("El argumento debe ser entero");
        } catch (Exception e) {
            e.printStackTrace();                              // RTE: Saldo insuficiente
        } finally {
            System.out.println("Estado final: "+caja);    // Estado final: Caja[C, 700.0]
        }                                                     // La Caja queda CERRADA
    }
}
```

Captura de excepciones. Otro Ejemplo

- Si se accede a un elemento de un array que no existe, el sistema lanza `IndexOutOfBoundsException`.
- Si se intenta convertir un `String` a `int` incorrecto, el sistema lanza `NumberFormatException`.

```
public class Test {
    public static void main(String[] args) {
        try {
            int bb = Integer.parseInt(args[0]);
            int bn = Integer.parseInt(args[1]);
            // ...
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Uso: Test <Int> <Int>");
            e.printStackTrace();
        } catch (NumberFormatException e) {
            System.out.println("Los argumentos deben ser enteros");
            e.printStackTrace();
        }
    }
}
```

Ejemplo del Flujo de Ejecución (I)

- Ejemplo si ocurre una excepción `ArrayIndexOutOfBoundsException`:

```
public class Test {
    public static void main(String[] args) {
        try {
            int bb = Integer.parseInt(args[0]); // Lanza ArrayIndexOutOfBoundsException
            int bn = Integer.parseInt(args[1]);
            int d = bb/bn;
        } catch (IndexOutOfBoundsException e) { // Captura la excepción
            System.out.println("Uso: Test <Int> <Int>");
            System.out.println(e);
        } catch (NumberFormatException e) {
            System.out.println("Los argumentos deben ser enteros");
            System.out.println(e);
        } finally {                                  // Se ejecuta siempre
            System.out.println("El programa sigue aquí");
        }
        System.out.println("Y después por aquí"); // El programa continua
    }
}
```

```
Uso: Test <Int> <Int>
java.lang.ArrayIndexOutOfBoundsException: 0
El programa sigue aquí
Y después por aquí
```

Ejemplo del Flujo de Ejecución (II)

- Ejemplo si ocurre una excepción `NumberFormatException`:

```
public class Test {
    public static void main(String[] args) {
        try {
            int bb = Integer.parseInt(args[0]); // Lanza NumberFormatException
            int bn = Integer.parseInt(args[1]);
            int d = bb/bn;
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Uso: Test <Int> <Int>");
            System.out.println(e);
        } catch (NumberFormatException e) {     // Captura la excepción
            System.out.println("Los args deben ser enteros");
            System.out.println(e);
        } finally {                             // Se ejecuta siempre
            System.out.println("El programa sigue aquí");
        }
        System.out.println("Y después por aquí"); // El programa continua
    }
}
```

```
Los argumentos deben ser enteros
java.lang.NumberFormatException: For input string: "20hola"
El programa sigue aquí
Y después por aquí
```

Ejemplo del Flujo de Ejecución (III)

- Ejemplo si el bloque `try` termina normalmente:

```
public class Test {
    public static void main(String[] args) {
        try {
            int bb = Integer.parseInt(args[0]);
            int bn = Integer.parseInt(args[1]);
            int d = bb/bn;
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Uso: Test <Int> <Int>");
            System.out.println(e);
        } catch (NumberFormatException e) {
            System.out.println("Los argumentos deben ser enteros");
            System.out.println(e);
        } finally {                                  // Se ejecuta siempre
            System.out.println("El programa sigue aquí");
        }
        System.out.println("Y después por aquí"); // El programa continua
    }
}
```

```
El programa sigue aquí
Y después por aquí
```

Ejemplo del Flujo de Ejecución (IV)

- Ejemplo si ocurre una excepción `ArithmeticException`:

```
public class Test {
    public static void main(String[] args) {
        try {
            int bb = Integer.parseInt(args[0]);
            int bn = Integer.parseInt(args[1]);
            int d = bb/bn;                              // Lanza ArithmeticException
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Uso: Test <Int> <Int>");
            System.out.println(e);
        } catch (NumberFormatException e) {
            System.out.println("Los args deben ser enteros");
            System.out.println(e);
        } finally {                                     // Se ejecuta siempre
            System.out.println("El programa sigue aquí");
        }
        System.out.println("Y después por aquí"); // La excepción no se captura
    }                                             // entonces se propaga
}
```

```
El programa sigue aquí
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main4.main(Main4.java:6)
```

Ejemplo del Flujo de Ejecución (V)

- Ejemplo si ocurre una excepción `ArithmeticException`:

```
public class DPC {
    private static int division(int num1, int num2) {
        int cociente;
        try {
            cociente =  num1 / num2;
        } finally {
            System.out.println("Finally hecho.");
        }
        System.out.println("Volviendo de división.");
        return cociente;
    }
    public static void main(String[] args) {
        System.out.println(division(10, 0));
        System.out.println("Volviendo de main.");
    }
}
```

```
Finally hecho.
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at DPC.division(DPC.java:5)
    at DPC.main(DPC.java:12)
```

Ejemplo del Flujo de Ejecución (VI)

- Ejemplo si ocurre una excepción `ArithmeticException`:

```
public class DPC {
    private static int division(int num1, int num2) {
        int cociente;
        try {
            cociente =  num1 / num2;
        } finally {
            System.out.println("Finally hecho.");
        }
        System.out.println("Volviendo de división.");
        return cociente;
    }
    public static void main(String[] args) {
        try {
            System.out.println(division(10, 0));
        } catch (ArithmeticException e) {
            System.out.println(e);
        }
        System.out.println("Volviendo de main.");
    }
}
```

```
Finally hecho.
java.lang.ArithmeticException: / by zero
Volviendo de main.
```

La sentencia *try-con-recursos*

- La sentencia **try-con-recursos** permite crear objetos *"auto-closeables"* (`PrintWriter`, `BufferedReader`, `Scanner`, etc) de forma controlada, de tal forma que se garantiza que se cerrarán automáticamente, incluso en presencia de excepciones.
  - Los recursos sólo son accesibles desde el bloque `try`.
  - Se invoca automáticamente al método `close()` al terminar el bloque `try` (antes del `catch`/`finally`).
  - Se pueden crear múltiples objetos *"auto-closeables"* separados por ;

```
import java.io.PrintWriter;
import java.io.IOException;
public class Ejemplo {
    private static final String[] DATOS = { "hola", "como", "estás" };
    private static void imprimirDatos(PrintWriter pw) {
        for (String pal : DATOS) {
            pw.println(pal);
        }
    }
    public static void main(String[] args) {
        try (PrintWriter pw = new PrintWriter(args[0])) {
            imprimirDatos(pw);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ERROR: falta el nombre del fichero");
        } catch (IOException e) {
            System.out.println("ERROR: no se puede escribir en el fichero");
        }
    }
}
```

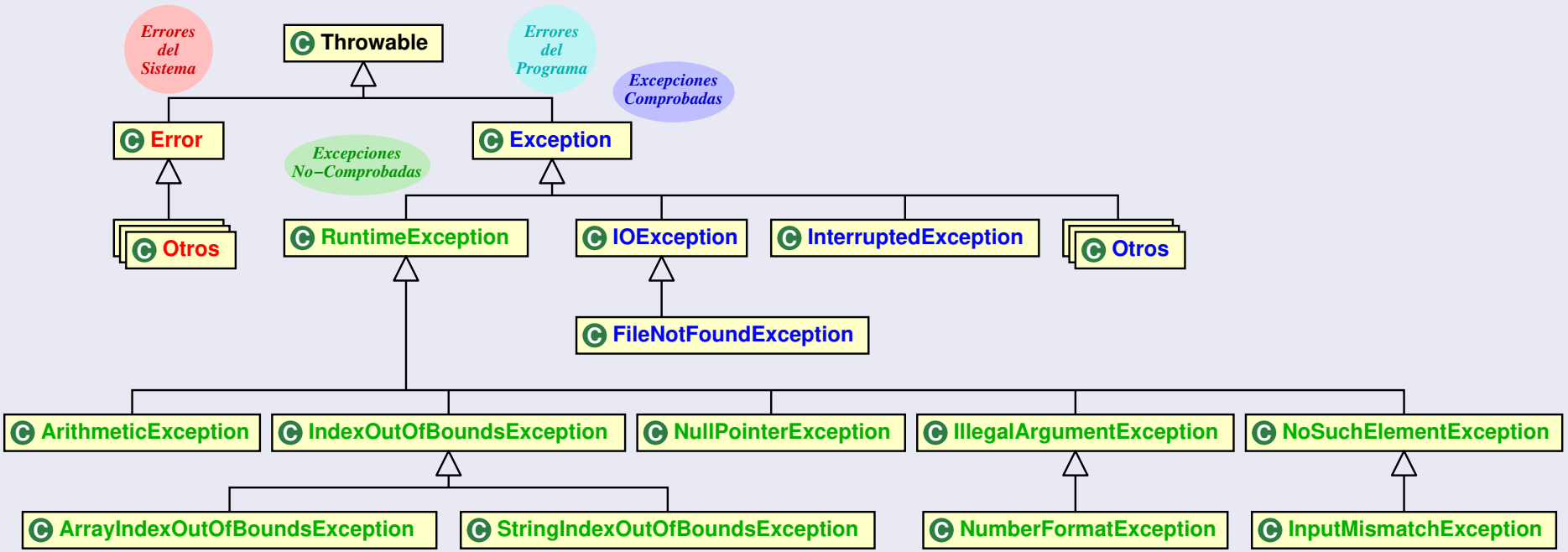
Equivalencia de la sentencia *try-con-recursos*

- La sentencia **try-con-recursos** sería equivalente a una sentencia `try` con bloque `finally` para cerrar los recursos.

```
import java.io.PrintWriter;
import java.io.IOException;
public class Ejemplo {
    private static final String[] DATOS = { "hola", "como", "estás" };
    private static void imprimirDatos(PrintWriter pw) {
        for (String pal : DATOS) {
            pw.println(pal);
        }
    }
    public static void main(String[] args) {
        PrintWriter pw = null;
        try {
            pw = new PrintWriter(args[0]);
            imprimirDatos(pw);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ERROR: falta el nombre del fichero");
        } catch (IOException e) {
            System.out.println("ERROR: no se puede escribir en el fichero");
        } finally {
            if (pw != null) {
                pw.close();
            }
        }
    }
}
```

Excepciones comprobadas y no comprobadas

- **Excepciones no comprobadas**: representan condiciones que, en términos generales, reflejan **errores de programación**, de los que no es posible recuperarse adecuadamente.
- **Excepciones comprobadas**: representan condiciones que, aunque excepcionales, es razonable que ocurran, y por consiguiente, deben ser **capturadas** o **anunciadas**.



Tratamiento de excepciones comprobadas

- Las **excepciones comprobadas** deben ser **capturadas** o **anunciadas**.
  - Capturadas: se **define** un manejador `catch` para ellas.
  - Anunciadas: se **anuncia** en la **cabecera** del método que puede **lanzar** excepciones de las clases especificadas (cláusula `throws`).
    - Aportan **información** muy importante al **usuario del método**, que deberán **capturarlas** o, en otro caso, **anunciarlas** también.

```
public void escrFich1(String nombre) {
    try (PrintWriter pw = new PrintWriter(nombre)) {
        pw.println("hola como estás");
    } catch (IOException e) {                              // Excep Capturada
        System.out.println("ERROR: no se puede escribir en el fichero");
    }
}
// ...
public void escrFich2(String nombre) throws IOException { // Excep Anunciada
    try (PrintWriter pw = new PrintWriter(nombre)) {
        pw.println("hola como estás");
    }
}
```

Tratamiento de excepciones comprobadas

```
public class EjemploExcepcionComprobada {
    public void metodoLanza() throws Exception {     // anuncia excepción comprobada
        if (error) {
            throw new Exception("Error");            // lanza excepción comprobada
        }
        // ...
    }
    public void metodoPropaga() throws Exception {   // anuncia excepción comprobada
        metodoLanza();                               // propaga excepción comprobada
        // ...
    }
    public void metodoCaptura() {
        try {
            metodoLanza();
            // ...
        } catch (Exception e) {                      // captura excepción comprobada
            // ...
        }
    }
}
```

Tratamiento de excepciones comprobadas. Ejemplo (I)

- Calcular la media y desviación típica de una colección aleatoria de números.

```
import java.util.Arrays;
public class Datos {
    private int[] datos;
    public Datos() {
        datos = new int[(int)(10 * Math.random())]; // puede ser cero
        for (int i = 0; i < datos.length; ++i) {
            datos[i] = (int)(100 * Math.random());
        }
    }
    public double media() throws Exception { // ANUNCIA LA EXCEPCIÓN LANZADA
        if (datos.length == 0) { throw new Exception("coleccion vacia"); }
        double suma = 0;
        for (int i = 0; i < datos.length; ++i) {
            suma += datos[i];
        }
        return suma / datos.length;
    }
    public double desvTipica() throws Exception { // ANUNCIA LA EXCEPCIÓN PROPAGADA
        double m = media();
        double suma = 0;
        for (int i = 0; i < datos.length; ++i) {
            suma += Math.pow(datos[i] - m, 2);
        }
        return Math.sqrt(suma / datos.length);
    }
    public String toString() { return Arrays.toString(datos); }
}
```

Tratamiento de excepciones comprobadas. Ejemplo (II)

- Generar una colección aleatoria de números. En caso de datos incorrectos (colección vacía o desviación típica mayor que 1) volver a generar la colección.

```
public class Main {
    public static void main(String[] args) {
        boolean ok = false;
        Datos d = null;
        do {
            try {                                   // VIGILA LAS EXCEPCIONES
                System.out.println("Generando datos ...");
                d = new Datos();
                double dt = d.desvTipica();  // Anuncia la excepción Exception
                ok = (0 <= dt && dt <= 1);
            } catch (Exception e) {                 // CAPTURA LA EXCEPCIÓN ANUNCIADA
                ok = false;
            }
        } while (! ok);
        System.out.println("Datos: "+d);      // datos, media y desviación típica
    }
}
```

Redefinición de métodos con cláusula `throws`

- La **redefinición** de un método en la **subclase** puede **especificar todas o un subconjunto** de las clases de excepciones (incluidas sus subclases) especificadas en la cláusula `throws` del método definido en la **superclase**.

```
public class A {
    // ...
    public void metodoX() throws Excepcion1, Excepcion2, Excepcion3 {
        // ...
    }
    // ...
}
public class B extends A {
    // ...
    public void metodoX() throws Excepcion1, Subc1Excepcion3 {
        // ...
    }
    // ...
}
```

Definiendo nuestras propias excepciones

- El programador puede **definir nuevas excepciones** como subclases de excepciones previamente definidas en el sistema.
  - `Exception` para excepciones **comprobadas**.
  - `RuntimeException` para excepciones **no comprobadas**.
  - Se deben definir, al menos, un constructor sin parámetros, y otro constructor con un único parámetro de tipo `String`.

Definir nueva excepción comprobada

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

Definir nueva excepción no-comprobada

```
public class DtsException extends RuntimeException {
    public DtsException() {
        super();
    }
    public DtsException(String msg) {
        super(msg);
    }
}
```

- La nueva excepción definida puede lanzarse como las demás excepciones.

```
throw new DtsException("Colección Vacía");
```

- La variable `serialVersionUID` es opcional (nosotros no la utilizaremos), y la puede generar automáticamente el entorno de programación (Eclipse).

Definiendo nuestras propias excepciones. Ejemplo (I)

- Calcular la media y desviación típica de una colección aleatoria de números.

```
import java.util.Arrays;
public class Datos {
    private int[] datos;
    public Datos() {
        datos = new int[(int)(10 * Math.random())]; // puede ser cero
        for (int i = 0; i < datos.length; ++i) {
            datos[i] = (int)(100 * Math.random());
        }
    }
    public double media() throws DtsException { // ANUNCIA LA EXCEPCIÓN LANZADA
        if (datos.length == 0) { throw new DtsException("Colección Vacía"); }
        double suma = 0;
        for (int i = 0; i < datos.length; ++i) {
            suma += datos[i];
        }
        return suma / datos.length;
    }
    public double desvTipica() throws DtsException { // ANUNCIA LA EXCEPCIÓN PROPAGADA
        double m = media();
        double suma = 0;
        for (int i = 0; i < datos.length; ++i) {
            suma += Math.pow(datos[i] - m, 2);
        }
        return Math.sqrt(suma / datos.length);
    }
    public String toString() { return Arrays.toString(datos); }
}
```

Definiendo nuestras propias excepciones. Ejemplo (II)

- Generar una colección aleatoria de números. En caso de datos incorrectos (colección vacía o desviación típica mayor que 1) volver a generar la colección.

```
public class Main {
    public static void main(String[] args) {
        boolean ok = false;
        Datos d = null;
        do {
            try {                                   // VIGILA LAS EXCEPCIONES
                System.out.println("Generando datos ...");
                d = new Datos();
                double dt = d.desvTipica();    // Anuncia la excepción DtsException
                ok = (0 <= dt && dt <= 1);
            } catch (DtsException e) {       // CAPTURA LA EXCEPCIÓN ANUNCIADA
                ok = false;
            }
        } while (! ok);
        System.out.println("Datos: "+d);     // datos, media y desviación típica
    }
}
```

Captura de una excepción y lanzamiento de otra excepción

- A veces resulta interesante capturar un tipo de excepción y lanzar otro tipo de excepción. Por ejemplo:

```
import java.util.Arrays;
public class Datos {
    // ...
    public double media() throws DtsException {
        try {
            double suma = 0;
            for (int i = 0; i < datos.length; ++i) {
                suma += datos[i];
            }
            return suma / datos.length;
        } catch (ArithmeticException e) {
            throw new DtsException("Colección Vacía");
            // throw new DtsException(e.getMessage());
        }
    }
    // ...
}
```

Reglas ante situaciones excepcionales: Preventiva

- **Preventiva**:
  - En aquellas situaciones donde se especifican las **precondiciones** requeridas para la invocación a determinados métodos. La violación de las precondiciones se suele indicar con excepciones *no comprobadas*, y suele indicar un error de programación.
  - Cuando la **comprobación** para evitar la situación excepcional es **poco costosa** y hay situaciones habituales en las que es **probable** que se produzca la excepción.
  - Entonces, se suele introducir código para **evitar** que la situación errónea excepcional pueda suceder.
  - Por ejemplo:

```
public void xxx(String n, double s) {
    // ...
    if (s >= 0) {
        banco.abrirCuenta(n, s);
    } else {
        // ...
    }
}
```

Reglas ante situaciones excepcionales: Curativa

- **Curativa**:
  - Cuando la **comprobación** para evitar la situación excepcional es **costosa** y es **raro** encontrar situaciones donde se puede producir la excepción.
  - Entonces, **no** se suele **evitar** que la situación errónea excepcional pueda suceder, y se prepara el código para su manejo de forma adecuada.
  - Ejemplo:

```
public void xxx(String str) throws NumberFormatException {
    // ...
    int n = Integer.parseInt(str);
    // ...
}
public void zzz(String str) {
    try {
        // ...
        int n = Integer.parseInt(str);
        // ...
    } catch (NumberFormatException e) {
        // ...
    }
}
```

Ejemplo (I)

- En el paquete `banco`, defina la excepción **comprobada** `CuentaException` (derivada de `Exception`), así como las excepciones `SaldoNegativoException`, y `SaldoInsuficienteException`, derivadas de `CuentaException`.
- En el paquete `banco`, defina la clase `Cuenta` con los siguientes constructores y métodos públicos, considerando que una cuenta almacena el nombre del titular, su número de cuenta, y el saldo:
  - `Cuenta(String t, int n, double s);`
    - Lanza la excepción `SaldoNegativoException` si el saldo inicial es negativo.
    - En otro caso, crea una cuenta con los parámetros suministrados.
  - `void movimiento(double valor);`
    - Si el resultado de la suma del saldo actual con el valor recibido como parámetro es negativa, entonces lanza la excepción `SaldoInsuficienteException`.
    - En otro caso, incrementa o decrementa el valor del saldo actual con el valor recibido como parámetro.
  - `String toString();` Devuelve la representación textual de la cuenta.
- El programa principal recibe como argumentos el nombre del titular de la cuenta, el número de cuenta y el saldo inicial, y creará una cuenta. Los siguientes argumentos proporcionan las cantidades a ingresar y extraer.
  - En caso de error por *saldo inicial negativo*, se creará la cuenta con saldo inicial cero, y se continuará con la ejecución del resto del programa.
  - En caso de cualquier otro error, finalizará el programa con un mensaje adecuado.

Ejemplo (II)

```java
// File: banco/CuentaException.java
package banco;
public class CuentaException extends Exception {                    // excepción COMPROBADA
    public CuentaException() { super(); }
    public CuentaException(String s) { super(s); }
}

// File: banco/SaldoNegativoException.java
package banco;
public class SaldoNegativoException extends CuentaException {       // excepción COMPROBADA
    public SaldoNegativoException() { super(); }
    public SaldoNegativoException(String s) { super(s); }
}

// File: banco/SaldoInsuficienteException.java
package banco;
public class SaldoInsuficienteException extends CuentaException {  // excepción COMPROBADA
    public SaldoInsuficienteException() { super(); }
    public SaldoInsuficienteException(String s) { super(s); }
}
```

Ejemplo (III)

```java
// File: banco/Cuenta.java
package banco;
public class Cuenta {
    private String titular;
    private int numero;
    private double saldo;
    public Cuenta(String t, int n, double s) throws SaldoNegativoException {
        if (s < 0) {
            throw new SaldoNegativoException("Saldo inicial negativo");
        }
        titular = t;
        numero = n;
        saldo = s;
    }
    public void movimiento(double valor) throws SaldoInsuficienteException {
        if (saldo + valor < 0) {
            throw new SaldoInsuficienteException("Saldo insuficiente");
        }
        saldo += valor;
    }
    @Override
    public String toString() {
        return "[ " + titular + " " + numero + " " + saldo + " ]";
    }
}
```

Ejemplo (IV)

```java
// File: Main.java
import banco.*;
import java.util.Arrays;
public class Main {
    private static void procesarMovimientos(Cuenta c, String[] movs) throws SaldoInsuficienteException {
        for (int i = 0; i < movs.length; ++i) {
            c.movimiento(Double.parseDouble(movs[i]));                    // puede lanzar SaldoInsuficiente, que será p
        }
    }
    public static void main(String[] args) {
        try {
            Cuenta c;
            try {
                c = new Cuenta(args[0], Integer.parseInt(args[1]), Double.parseDouble(args[2]));
            } catch (SaldoNegativoException e) {                          // recuperación de error en caso de saldo ini
                c = new Cuenta(args[0], Integer.parseInt(args[1]), 0.0);
            }
            procesarMovimientos(c, Arrays.copyOfRange(args, 3, args.length));
            System.out.println("Estado: " + c);
        } catch (CuentaException e) {                                     // captura de cualquier excepción relacionada
            System.out.println("Error: "+e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {                      // captura de error de acceso con índice fuer
            System.out.println("Error: no hay argumentos suficientes al programa");
        } catch (NumberFormatException e) {                               // captura de error en conversión númerica
            System.out.println("Error de formato numérico"+e.getMessage());
        } catch (Exception e) {                                           // captura cualquier excepción
            System.out.println("Error inesperado");
            e.printStackTrace();
        } finally {
            System.out.println("Fin del programa");                       // finally se ejecuta siempre
        }
    }
}
```