# Tratamiento de errores, excepciones

# Contenido

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

# El concepto de “excepción”

- Durante la ejecución de un programa se pueden producir errores de diversos niveles de severidad:
  - un índice fuera de rango,
  - una división por cero.
  - un fichero que no puede encontrarse o no existe,
  - un enlace de red que falla,
  - ...

# El concepto de “excepción”

- El código se volvería ininteligible si continuamente tuviéramos que comprobar las posibles condiciones de error sentencia tras sentencia.

# El concepto de “excepción”

- Consideremos el (pseudo)código del siguiente método que lee un fichero y copia su contenido en memoria.

¿Qué pasa si el fichero no puede abrirse?

¿Qué pasa si no puede determinarse la longitud del fichero?

```
leerFichero() {
    abrir el fichero;
    determinar la longitud del fichero;
    reservar la memoria suficiente;
    copiar el fichero en memoria;
    cerrar el fichero;
}
```

¿Qué pasa si no puede reservarse memoria suficiente?

¿Qué pasa si falla la lectura?

¿Qué pasa si el fichero no puede cerrarse?

# El concepto de “excepción”

```
TipoDeCódigoDeError leerFichero() {
   TipoDeCódigoDeError códigoDeError = 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) { códigoDeError = -1; }
         } else { códigoDeError = -2; }
      } else { códigoDeError = -3; }
      cerrar el fichero;
      if (códigoDeError == 0 && el fichero no se cerró){
         códigoDeError = -4;
      }
   } else { códigoDeError = -5; }
   return códigoDeError ;
}
```

# El concepto de “excepción”

```
TipoDeCódigoDeError leerFichero() {
   TipoDeCódigoDeError códigoDeError = 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) { códigoDeError = -1; }
         } else { códigoDeError = -2; }
      } else { códigoDeError = -3; }
      cerrar el fichero;
      if (códigoDeError == 0 && el fichero no se cerró){
         códigoDeError = -4;
      }
   } else { códigoDeError = -5; }
   return códigoDeE
}
```

- Difícil de leer
- Se pierde el flujo lógico de ejecución
- Difícil de modificar

# El concepto de “excepción”

- El mecanismo de las excepciones proporciona una forma clara de comprobar posibles errores sin oscurecer el código.

# El concepto de “excepción”

```
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”

```
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
        ...;
    }
}
```

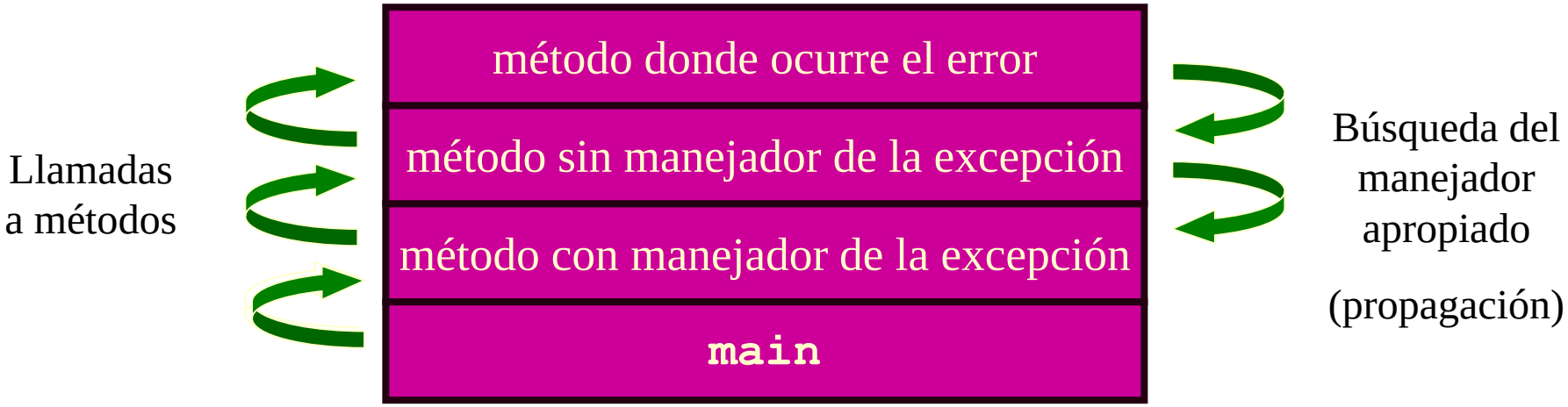
Las excepciones no nos liberan de hacer la detección, de informar y de manejar los errores, pero nos permiten escribir el flujo principal de nuestro código en un sitio y de tratar los casos excepcionales separadamente.

# El concepto de “excepción”

- Una *excepción* es un objeto que representa un evento (una señal) que interrumpe el flujo normal de instrucciones durante la ejecución de un método como consecuencia de un error (condición excepcional).
  - Decimos que se *lanza* una excepción (bien por el sistema o por el propio programador (throw)).
- La excepción puede ser *capturada* para su tratamiento (manejo).
- Si no es tratada, el programa termina (mostrando información detallada sobre ella).

# Propagación de excepciones

- Cuando se lanza una excepción, el sistema de ejecución recorre la "pila de llamadas a métodos" en orden inverso buscando un método que contenga un bloque de código que capture la excepción (*manejador de excepción*). La excepción se va propagando.




Pila de llamadas a métodos

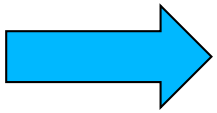
- Un método puede lanzar y capturar (manejar) una excepción
- Un método puede capturar un tipo de excepción y lanzar otro tipo de excepción.

# Lanzar una excepción

- Para lanzar una excepción (por el programador) se utiliza la instrucción:

```
throw <objeto-excepción>;
```

Ejemplo. En una clase **`Banco`** tenemos el siguiente código:



```
public class Banco {
    ...
    public int abrirCuenta(String titular, double saldo) {
        if (saldo < 0) {
            throw new RuntimeException( "Saldo negativo" );
        }
        ...
    }
}
```

¡Nos piden abrir una cuenta con saldo negativo!

No sabemos qué hacer. Lanzamos la excepción

# Lanzar una excepción

```
public class TestBanco {
    public static void main(String[] args) {
        Banco b = new Banco("Bueno");
        int nCta = b.abrirCuenta("Antonio",-500);
        …
    }
}
```

Como no capturamos la excepción, el programa termina con:

```
Exception in thread "main" java.lang.RuntimeException: Saldo negativo
at Banco.abrirCuenta(Banco.java:26)
at TestBanco.main(TestBanco.java:5)
```

# Lanzar una excepción

- Otro ejemplo (el sistema lanza la excepción): Supongamos una aplicación que crea una ***Urna*** con un número determinado de bolas blancas y otro de bolas negras y, posteriormente, realiza un proceso con ella. El número de bolas blancas y negras se pasan como argumentos al método **main**:

```
public class TestUrna {
    public static void main(String [] args) {
        int bb = Integer.parseInt(args[0]);
        int bn = Integer.parseInt(args[1]);
        ...
    }
}
```

Si **`args.length < 2`** el sistema lanza **`ArrayIndexOutOfBoundsException`**

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
at TestUrna.main(TestUrna.java:4)
```

# Lanzar una excepción

- Otro ejemplo (el sistema lanza la excepción): Supongamos una aplicación que crea una ***Urna*** con un número determinado de bolas blancas y otro de bolas negras y, posteriormente, realiza un proceso con ella. El número de bolas blancas y negras se pasan como argumentos al método **main**:

```
public class TestUrna {
    public static void main(String [] args) {
        int bb = Integer.parseInt(args[0]);
        int bn = Integer.parseInt(args[1]);
    }
}
```

Si **`args[0]`** o **`args[1]`** no es convertible a entero el método parseInt lanza una **`NumberFormatException`**

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "2e"
at java.lang.NumberFormatException.forInputString(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at TestUrna.main(TestUrna.java:4)
```

# Captura de excepciones
# La instrucción `try-catch`

```
try (
      // Se crean objetos “closeables” que se cerrarán
      // automáticamente al terminar el bloque try
) {
       //
        // cuerpo vigilado
       //
} catch (Excepción11 | Excepción12 | ... | Excepción1N e)  {

      // Tratamiento común para todas las excepciones capturadas

} catch (Excepción21 | Excepción22 | ... | Excepción2N e)  {

      // Tratamiento común para todas las excepciones capturadas

...

} catch (ExcepciónN1 | ExcepciónN2 | ... | ExcepciónNN e)  {

      // Tratamiento común para todas las excepciones capturadas

} finally {
       // código que se ejecuta siempre al final (haya o no excepciones)
}
```

# Captura de excepciones
# La instrucción `try-catch`

```
try (
      // Se crean objetos “closeables” que se cerrarán
      // automáticamente al terminar el bloque try
) {
      //
            uerpo vigilado
      //
} catc      cepción11 | Excepción12 | ... | Excepción1N e)  {

      //      iento común para todas las excepciones capturadas

} catc      ión21 | Excepción22 | ... | Excepción2N

                                    cepciones c

                                    xcepciónNN e)  {

                                    cepciones cap

} finally {
      // código que se ejecuta siempre al final (haya o n
}
```

El bloque **`try`** vigila código para capturar las posibles excepciones que se produzcan.

Los bloques **`catch`** se ejecutan si se lanza una excepción adecuada en el bloque vigilado (son opcionales)

Es posible abrir objetos closeables que se cerrarán automáticamente cuando termine el bloque **`try`** (es opcional)

El bloque **`finally`** se ejecuta siempre (es opcional)

# La captura de excepciones

```
public class TestUrna {
    public static void main(string [] args) {
      try {
            int bb = Integer.parseInt(args[0]);
            int bn = Integer.parseInt(args[1]);
            ...
      } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Uso: TestUrna <Int> <Int>");
      } catch (NumberFormatException ee) {
            System.err.println("Los args deben ser enteros");
      }
    }
}
```

Es usual utilizar err en lugar de out para los mensajes de error

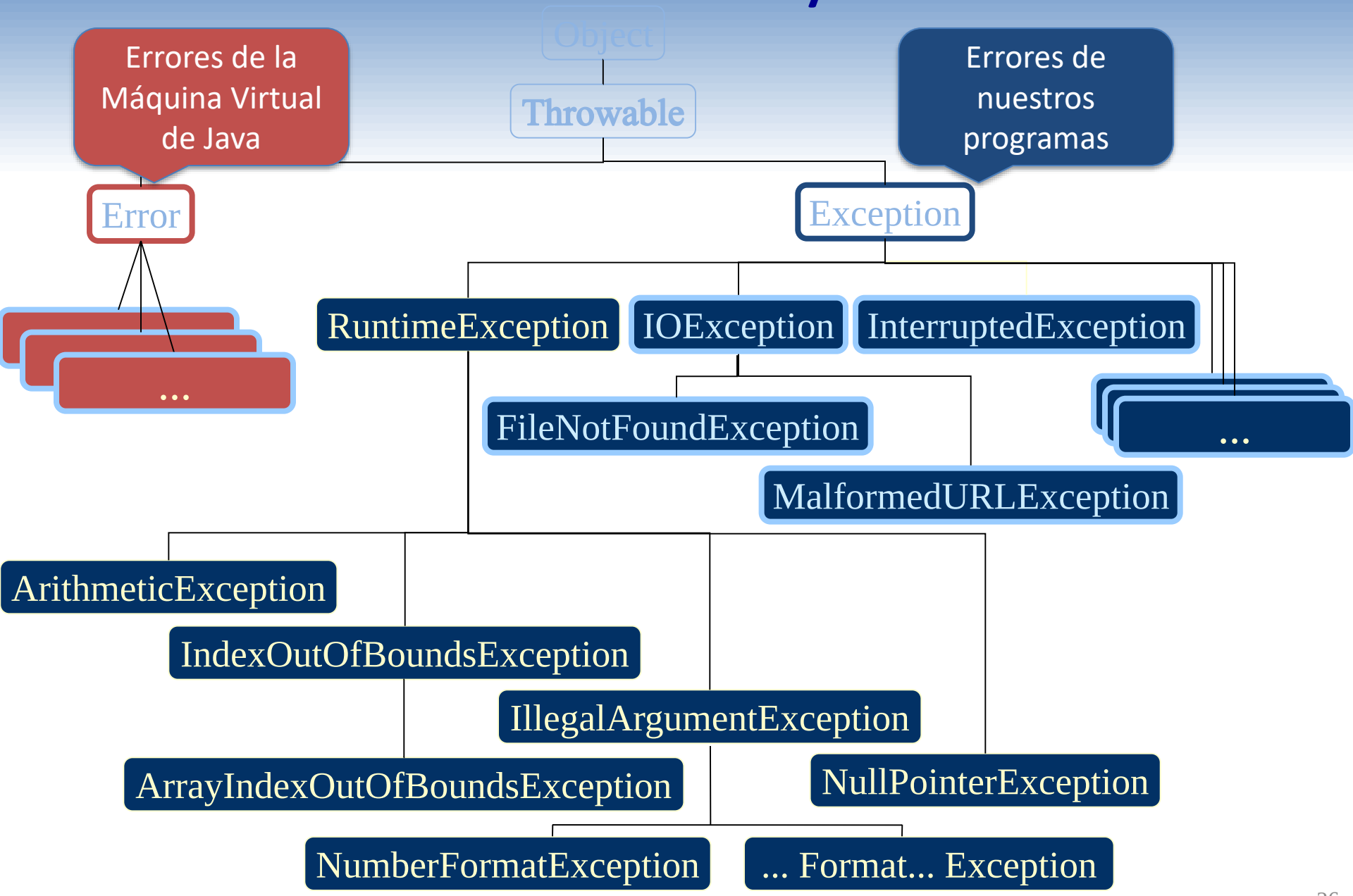
# La captura de excepciones

- Los manejadores de excepciones pueden hacer cosas diferentes a terminar el programa con un mensaje de error. Por ejemplo:
  - Preguntar al usuario por la decisión a tomar
  - Recuperarse del error y continuar con la ejecución del programa.
  - Lanzar otro tipo de excepción.

- Cuando una excepción es capturada (manejada) por un bloque **catch**, tras ejecutar el código del mismo (y el posible código del *finally* si existe), la ejecución del programa continúa por la siguiente instrucción al **try-catch**

- Si una excepción no es capturada (manejada) por un bloque **catch**, se ejecuta el posible código del *finally* si existe, el código que hubiera detrás del **try-catch** no se ejecuta, propagándose la excepción (en caso de que el código esté en el main, el programa terminará)

# La clase **Throwable** y sus subclases



# La clase **`Throwable`**

- Sólo pueden lanzarse objetos que son instancias de la clase **`Throwable`** (o de una de sus subclases).

- Por convenio, la clase **`Throwable`** y sus subclases tienen dos constructores:
  - uno sin argumentos
  - otro con un argumento de tipo String, el cual puede utilizarse para almacenar mensajes de error.

- Un objeto de la clase **`Throwable`** contiene el estado de la pila de llamadas en el momento en que fue creado.

# La clase **Throwable**

Entre los distintos métodos que tiene están:

**`String getMessage()`**

Devuelve el texto con el mensaje de error del objeto.

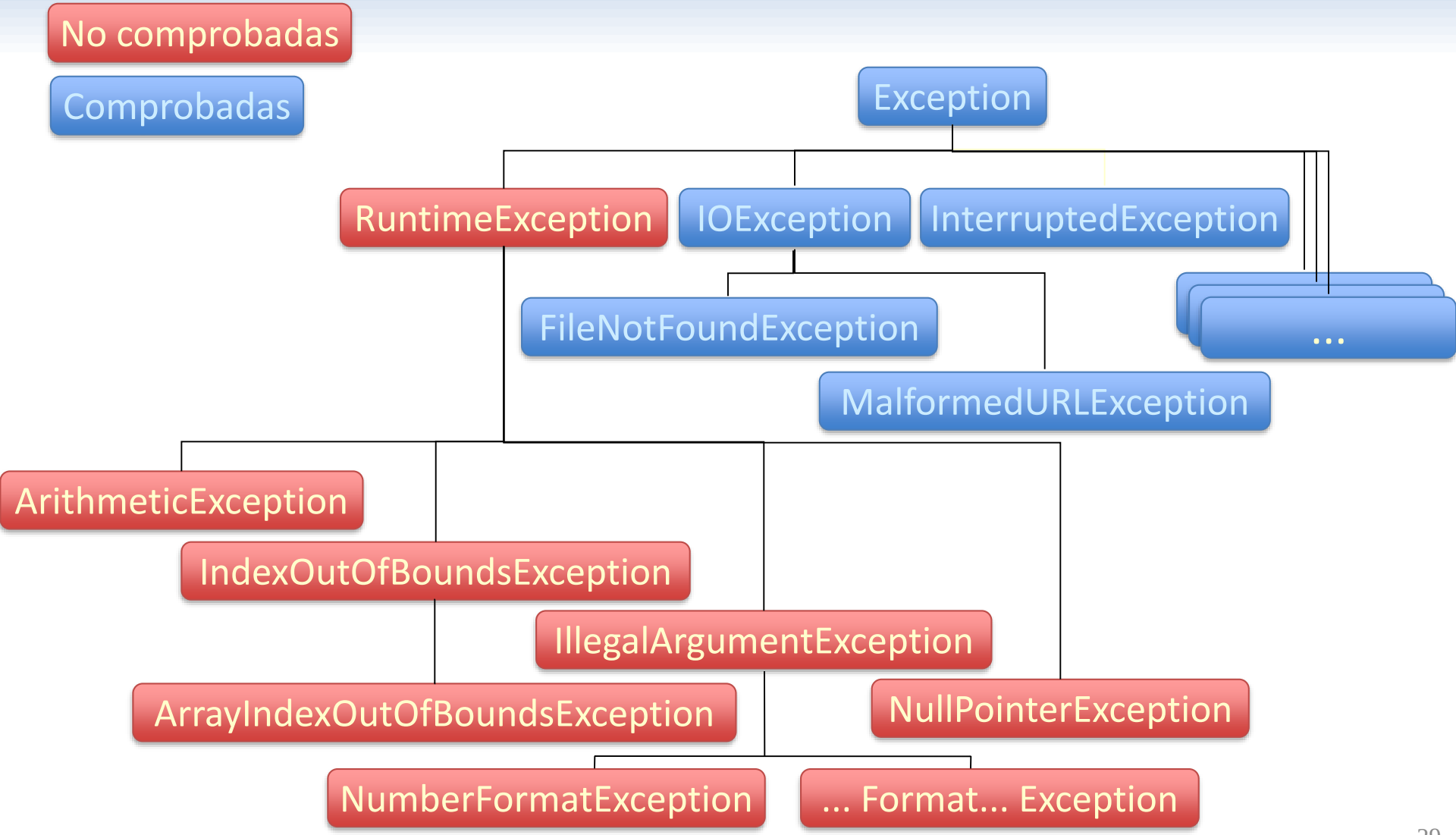
```
throw new RuntimeException( “Saldo negativo” );
```

**`void printStackTrace()`**

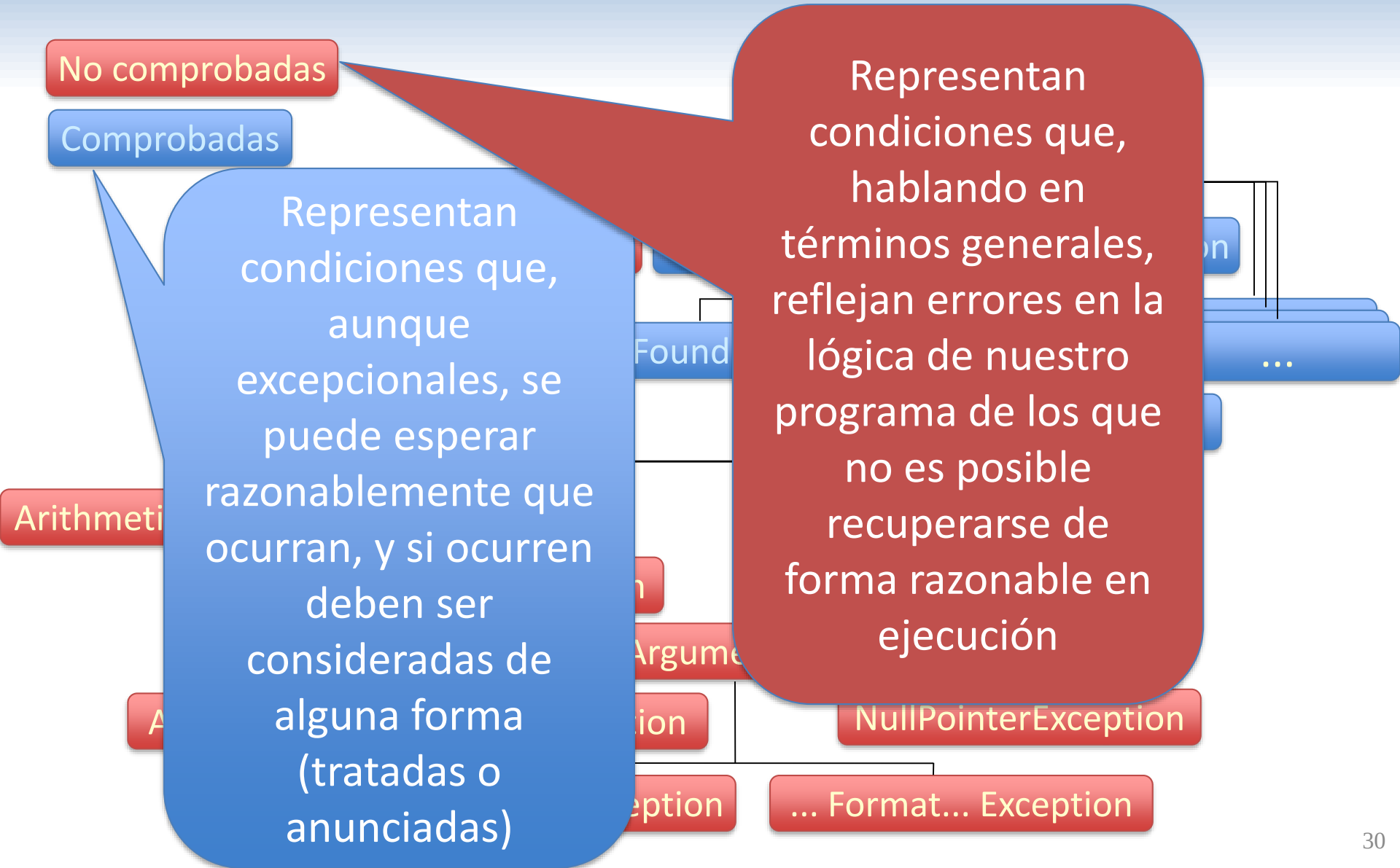
Imprime este objeto y su traza en la salida de errores estándar.

```
Exception in thread "main" java.lang.RuntimeException: Saldo negativo
at Banco.abrirCuenta(Banco.java:26)
at TestBanco.main(TestBanco.java:5)
```

# Excepciones comprobadas y no comprobadas (por el compilador)



# Excepciones comprobadas y no comprobadas (por el compilador)



# Tratamiento de excepciones comprobadas

- Las excepciones comprobadas deben ser capturadas o anunciadas
- Capturadas: se hace un tratamiento con ellas.
- Anunciadas: se anuncian en la cabecera del método (aportan información al usuario del método; tan importantes como el tipo de los parámetros y el tipo del valor devuelto)

```
public void espera(int millis) {
        try {
                Thread.sleep(millis);
        } catch (InterruptedException e) {
                System.err.println(”se interrumpió");
        }
}
```

Capturada

# Tratamiento de excepciones comprobadas

- Las excepciones comprobadas deben ser capturadas o anunciadas
- Capturadas: se hace un tratamiento con ellas.
- Anunciadas: se anuncian en la cabecera del método (aportan información al usuario del método; tan importantes como el tipo de los parámetros y el tipo del valor devuelto)

```
public void espera(int millis) throws InterruptedException {
        Thread.sleep(millis);
}
```

Anunciada

- Pueden anunciarse varias excepciones separadas por comas
- Las excepciones no comprobadas (ej. ArrayIndexOutOfBounds) si queremos las podemos anunciar también, aunque no es obligatorio

# Excepciones relacionadas

```
public class TestUrna {
    public static void main(String[] args) {
        try {
            int bb = Integer.parseInt(args[0]);
            int bn = Integer.parseInt(args[1]);
            ...
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Uso: TestUrna <Int> <Int>");
        } catch (NumberFormatException ee) {
            System.err.println("Los argumentos deben ser números enteros");
        } catch (RuntimeException e) {
            System.err.println(”Alguna excepción distinta");
        }
    }
}
```

correcto

# Excepciones relacionadas

```
public class TestUrna {
    public static void main(String[] args) {
        try {
            int bb = Integer.parseInt(args[0]);
            int bn = Integer.parseInt(args[1]);
            ...
        } catch (NumberFormatException ee) {
            System.err.println("Los argumentos deben ser números enteros");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Uso: TestUrna <Int> <Int>");
        } catch (RuntimeException e) {
            System.err.println(”Alguna excepción distinta");
        }
    }
}
```

correcto

# Excepciones relacionadas

```
public class TestUrna {
    public static void main(String[] args) {
        try {
            int bb = Integer.parseInt(args[0]);
            int bn = Integer.parseInt(args[1]);
            ...
        } catch (RuntimeException e) {
            System.err.println(”Alguna excepción distinta");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Uso: TestUrna <Int> <Int>");
        } catch (NumberFormatException ee) {
            System.err.println("Los argumentos deben ser números");
        }
    }
}
```

incorrecto

Inalcanzable porque hereda de RuntimeException

Inalcanzable porque hereda de RuntimeException

# Definiendo nuestras propias excepciones

Un usuario puede definir sus propias excepciones.

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

No Comprobada porque hereda de RuntimeException

- Y ahora puede lanzarse como las demás y puede o no ser tratada.

```
throw new MiExcepción("…");
```

# Ejemplo: lanzando y tratando nuestra excepción

```
public class Ejemplo {
    private static void división(int num1, int num2) {
      if (num2 == 0) {
            throw new MiExcepción("Intento de dividir por 0" );
      }
      System.out.println(num1 + " / " + num2 + " = " + (num1 / num2));
    }
    public static void main(String[] args) {
      try {
            división(10, 0);
            System.out.println("División hecha.");
      } catch (MiExcepción e) {
            System.err.println("ERROR: " + e.getMessage());
      }
    }
}
```

Se lanza

Se trata

**Salida:**

**ERROR: Intento de dividir por 0**

# Definiendo nuestras propias excepciones

Un usuario puede definir sus propias excepciones.

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

Comprobada porque hereda de Exception

- Y ahora puede lanzarse como las demás y debe ser tratada o anunciada.

```
throw new MiExcepción("…");
```

# Ejemplo: lanzando y tratando nuestra excepción

```
public class Ejemplo {
    private static void división(int num1, int num2) throws MiExcepción {
      if (num2 == 0) {
            throw new MiExcepción("Intento de dividir por 0" );
      }
      System.out.println(num1 + " / " + num2 + " = " + (num1 / num2));
    }
    public static void main(String[] args) {
      try {
            división(10, 0);
            System.out.println("División hecha.");
      } catch (MiExcepción e) {
            System.err.println("ERROR: " + e.getMessage());
      }
    }
}
```

Se anuncia

Se lanza

Se trata

**Salida:**

**ERROR: Intento de dividir por 0**