Tratamiento de excepciones en Java

El tratamiento de excepciones en Java es un mecanismo del lenguaje que permite gestionar errores y situaciones *excepcionales*.

1. Concepto de excepción

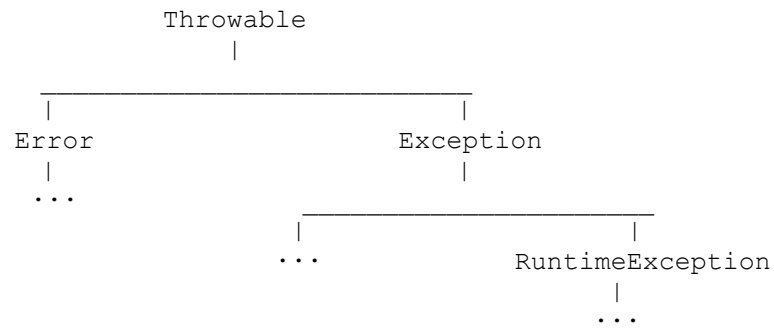
Una excepción en Java (así como en otros muchos lenguajes de programación) es un error o situación excepcional que se produce durante la ejecución de un programa. Algunos ejemplos de errores y situaciones excepcionales son:

- Leer un fichero que no existe
- Acceder al valor N de una colección que contiene menos de N elementos
- Enviar/recibir información por red mientras se produce una perdida de conectividad

Todas las excepciones en Java se representan a través de objetos que heredan, en última instancia, de la clase `java.lang.Throwable`.

2. Tipos de excepciones

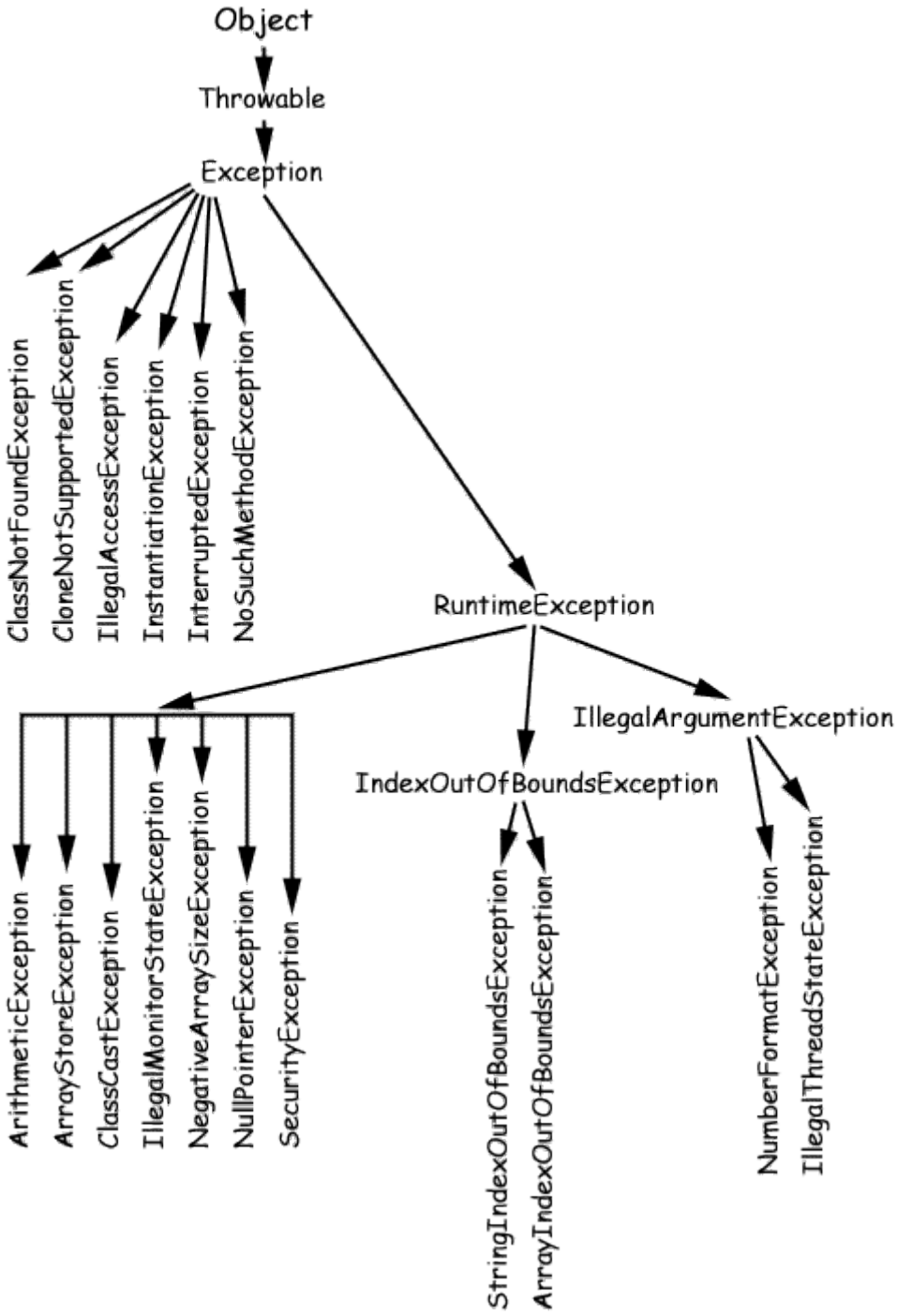
El lenguaje Java diferencia claramente entre tres tipos de excepciones: **errores**, comprobadas o verificadas (en adelante ***checked***) y no comprobadas o no verificadas (en adelante ***unchecked***). El gráfico que se muestra a continuación muestra el árbol de herencia de las excepciones en Java (se omite el paquete de todas las que aparecen, que es `java.lang`):



La clase principal de la cual heredan todas las excepciones Java es `Throwable`. De ella nacen dos ramas: `Error` y `Exception`. La primera representa errores de una magnitud tal que una aplicación nunca debería intentar realizar nada con ellos (como errores de la JVM, desbordamientos de buffer, etc). La segunda rama, encabezada por `Exception`,

representa aquellos errores que normalmente *si* solemos gestionar, y a los que comunmente solemos llamar *excepciones*.

De `Exception` nacen múltiples ramas: `ClassNotFoundException`, `IOException`, `ParseException`, `SQLException` y otras muchas, todas ellas de tipo checked. La única excepción a esto es `RuntimeException` que es de tipo unchecked y encabeza todas las de este tipo.



3. Excepciones checked

Una excepción de tipo checked representa un error del cual técnicamente podemos recuperarnos. Por ejemplo, una operación de lectura/escritura en disco puede fallar porque el fichero no exista, porque este se encuentre bloqueado por otra aplicación, etc. Todos estas situaciones, además de ser inherentes al propósito del código que las lanza (lectura/escritura en disco) son totalmente ajenas al propio código, y deben ser (y de hecho son) declaradas y manejadas mediante excepciones de tipo checked y sus mecanismos de control.

En ciertos momentos, a pesar de la promesa de *recuperabilidad*, nuestro código no estará preparado para gestionar la situación de error, o simplemente no será su responsabilidad. En estos casos lo más razonable es relanzar la excepción y confiar en que un método superior en la cadena de llamadas sepa gestionarla.

Por tanto, todas las excepciones de tipo checked deben ser **capturadas** o **relanzadas**.

En el primer caso, utilizamos el bloque `try-catch`. Ej:

```
import java.io.FileWriter;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        FileWriter fichero;
        try {
            // Las siguientes dos líneas pueden lanzar una excepción
de tipo IOException
            fichero = new FileWriter("ruta");
            fichero.write("Esto se escribirá en el fichero");
        } catch (IOException ioex) {
            // Aquí capturamos cualquier excepción IOException que se
lance (incluidas sus subclases)
            ioex.printStackTrace();
        }

    }
}
```

En caso de querer relanzar la excepción, debemos declarar dicha intención en la firma o prototipo del método que contiene las sentencias que lanzan la excepción, y lo hacemos mediante la claúsula **throws**:

```
import java.io.FileWriter;
import java.io.IOException;

public class Main {
    // En lugar de capturar una posible excepción, la relanzamos
    public static void main(String[] args) throws IOException {
        FileWriter fichero = new FileWriter("ruta");
        fichero.write("Esto se escribirá en el fichero");
    }
}
```

Hay que tener presente que cuando se relanza una excepción estamos forzando al código cliente de nuestro método a capturarla o relanzarla. Una excepción que sea relanzada una y otra vez *hacia arriba* terminará llegando al método primigenio y, en caso de no ser capturada por éste, producirá la finalización de su hilo de ejecución (thread). En el caso de aplicaciones de consola con un único hilo, como las que hemos estado viendo hasta ahora, ese método primigenio es el método **main**.

¿Cuándo capturar una excepción? ¿Cuándo relanzarla?

Capturamos una excepción cuando:

- Podemos recuperarnos del error y continuar con la ejecución
- Queremos registrar el error
- Queremos relanzar el error con un tipo de excepción distinto

En definitiva, cuando tenemos que realizar algún tratamiento del propio error. Por contra, relanzamos una excepción cuando:

- No es competencia nuestra ningún tratamiento de ningún tipo sobre el error que se ha producido

4. Excepciones unchecked

Una excepción de tipo unchecked representa un error de programación. Uno de los ejemplos más típicos es el de intentar leer en un array de N elementos un elemento que se encuentra en una posición N o mayor que N:

```
int[] numerosPrimos = {1, 3, 5, 7, 9, 11, 13, 17, 19, 23};    // Array
de diez elementos
int undecimoPrimo = numerosPrimos[10];    // Accedemos al undécimo
elto
```

El código anterior accede a una posición inexistente dentro del array, y su ejecución lanzará la excepción unchecked `ArrayIndexOutOfBoundsException` (excepción de índice de array fuera de límite). Esto es claramente un error de programación, ya que el código debería haber comprobado el tamaño del array antes de intentar acceder a una posición concreta:

```
int[] numerosPrimos = {1, 3, 5, 7, 9, 11, 13, 17, 19, 23};
int indiceUndecimoPrimo = 10;
if(indiceUndecimoPrimo > numerosPrimos.length) {
    System.out.println("El índice proporcionado (" +
indiceUndecimoPrimo + ") es mayor que el tamaño del array (" +
numerosPrimos.length + ")");
} else {
    int undecimoPrimo = numerosPrimos[indiceUndecimoPrimo];
    // ...
}
```

El aspecto más destacado de las excepciones de tipo unchecked es que no deben ser forzosamente declaradas ni capturadas (en otras palabras, no son verificadas). Por ello no son necesarios bloques `try-catch` ni declarar formalmente en la firma del método el lanzamiento de excepciones de este tipo. Esto, por supuesto, también afecta a métodos y/o clases más *hacia arriba* en la cadena invocante.

5. Creando nuestras propias excepciones

Aprovechando dos de las características más importantes de Java, la herencia y el polimorfismo, podemos crear nuestras propias excepciones de forma muy simple:

```
class CreditoInsuficienteException extends Exception {
    // ...
}
```

La clase del código anterior extiende a `Exception` y por tanto representa una excepción de tipo checked. Tal como su nombre indica, sería lanzada cuando durante una operación comercial no exista suficiente crédito. Esta situación excepcional es inherente a nuestra transacción comercial y no se genera por un defecto de código (no es un error de programación), y por tanto debe gestionarse con anterioridad:

```
class CarritoDeLaCompra {
    // ...

    public void pagarCompraConTarjeta() {
        try {
            TarjetaDeCredito tarjetaPreferida =
cliente.getTarjetaDeCreditoPreferida();
            tarjetaPreferida.realizarPago(getImporteCompra());
            cliente.enviarEmailConfirmación();
        } catch(CreditoInsuficienteException ciex) {
            // Informamos al usuario de crédito insuficiente
        }
    }
}
```

Si por el contrario deseamos crear una excepción de tipo unchecked, debemos hacer que nuestra clase extienda de `RuntimeException`. Volviendo al último ejemplo, podríamos pensar que `CreditoInsuficienteException` podría ser declarada como una excepción de tipo unchecked, ya que *siempre* es posible validar el saldo de la tarjeta de credito con anterioridad al pago (como haciamos con los índices del array). Sin embargo esto no siempre es posible ni razonable ya que:

- No disponemos del código fuente, sólo somos clientes de una librería escrita por terceros

- Aunque dispusiéramos del código fuente, no deberíamos estar autorizados a conocer el crédito disponible de ningún cliente (esto es información muy sensible y por tanto, confidencial)

6. Malas prácticas de uso

```
try {
    // Código que declara lanzar excepciónes
} catch(Exception ex) {}
```

El código anterior ignorará cualquier excepción que se lance dentro del bloque `try`, o mejor dicho, capturará toda excepción lanzada dentro del bloque `try` pero la silenciará no haciendo nada (frustrando así el principal propósito de la gestión de excepciones checked: *gestiónala o relánzala*). Cualquier error de diseño, de programación o de funcionamiento en esas lineas de código pasará inadvertido tanto para el programador como para el usuario. Lo mínimamente aceptable dentro de un bloque `catch` es un mensaje de **log** informando del error:

```
try {
    // Código que declara lanzar excepciónes
} catch(Exception ex) {
    logging.log("Se ha producido el siguiente error: " +
ex.getMessage());
    logging.log("Se continua la ejecución");
}
```

O bien para el caso de las aplicaciones que estamos desarrollando (aplicaciones simples de consola) imprimir el error en pantalla.

Otra opcióne sería pintar una traza completa del error mediante uno de los métodos informativos de `Throwable`:

```
try {
    // Código que declara lanzar excepciónes
} catch(Excepcion ex) {
    ex.printStackTrace();    // Podemos añadir cualquier tratamiento
adicional antes y/o después de esta línea
}
```

Otro abuso del mecanismo de tratamiento de excepciones es cuando se está intentando escribir código que mejore el rendimiento de la aplicación:

```
try {
    int i = 0;
    while(true) {
        System.out.println(numerosPrimos[i++]);
    }
} catch(ArrayIndexOutOfBoundsException aioobex) {}
```

El ejemplo anterior itera el array de números primos sin preocuparse de los límites del array (tal como haría de manera formal un bucle `for`) hasta sobrepasar el índice máximo, momento en el cual se lanzará una excepción de tipo `ArrayIndexOutOfBoundsException` que será capturada y silenciada. Esto es un error porque:

- El tratamiento de excepciones está diseñado para gestionar excepciones y no para realizar optimizaciones

- El código dentro de bloques try-catch no dispone de ciertas optimizaciones de las JVM más modernas (por ejemplo, y aplicable a nuestro caso, iteración de colecciones)

Otro error común se produce cuando estamos creando nuestra propia librería de excepciones y nos excedemos declarando excepciones checked. Las excepciones checked al contrario que los códigos `return` de lenguajes como C, fuerzan al programador a manejar condiciones excepcionales, mejorando así la legibilidad del código. Sin embargo, esta obligación puede llegar a *cargar* el código cliente:

```
try {
    // Código que declara lanzar muchas excepciónes
} catch(UnTipoDeException ex1) {
    // Gestionar...
} catch(OtroTipoDeException ex2) {
    // Gestionar...
} catch(OtroTipoMasDeException ex3) {
    // Gestionar...
} catch(OtroTipoTodaviaMasDeException ex3) {
    // Gestionar...
}
```

El código anterior suele abrumar, y el cliente acabará tentado por la siguiente alternativa:

```
try {
    // Código que declara lanzar muchas excepciónes
} catch(Exception ex) {
    // Gestionar cualquier excepcion, pues todas heredan de Exception
    // Perdemos la ventaja de gestionar condiciones excepcionales
concretas
}
```

Por ello, se debe pensar detenidamente si la excepción es de tipo checked o unchecked. Cualquier situación excepcional que deje la aplicación en un estado irrecuperable y/o no sea inherente al propósito del código que la produce debe ser declarada como una excepción de tipo unchecked.

La siguiente mala práctica que vamos a ver está íntimamente relacionada con la anterior, y es la de lanzar excepciones de forma genérica:

```
public void miMetodo() throws Exception {
    // Código que declara lanzar muchas excepciones.
    // Sin embargo, en la firma del método declaramos lanzar una única
super-clase de todas éllas
}
```

Los clientes del método no sabrán jamás con que condiciones especiales se pueden encontrar, y por tanto no podrán gestionarlas; no tendrán más remedio que informar del error y detener la ejecución.