Capítulo 13. Asertos y Excepciones

A. J. Pérez

Asertos

Un aserto es una instrucción basada en una expresión booleana, en un punto determinado del programa, que el programador sabe que siempre debe ser verdadera.

Ejemplos de asertos son las *precondiciones, los invariantes y las postcondiciones* de la *programación por contratos.*

La técnica de la programación con asertos consiste en distribuir por el código *chequeos de integridad*, de las condiciones que siempre deben cumplirse en un funcionamiento correcto. De esta forma si algo no es normal, se detecta lo antes posible, por sí mismo, en vez de tener que estar revisando trazas de ejecución buscando las causas que han llevado a un fallo.

Se puede decir que cuando un programador -mientras escribe código- piensa en dejar una traza de los valores que deben tener las variables en un punto, lo que necesita es un aserto.

Volviendo a la definición, se observa que si la expresión booleana -el aserto- es cierta, el sistema comprueba que el programa se está ejecutando dentro de los límites previstos; reduciendo la posibilidad de errores.

Los asertos no son nada nuevo en programación, de hecho los asertos ya estaban previstos en la especificación inicial de *oak* (el *lenguaje precursor de Java*) pero fueron desestimados porque no había tiempo suficiente para hacer una implementación satisfactoria.

Uso

Para declarar un aserto en un método, se usa la palabra clave **`assert`** que tiene dos sintaxis:

```
assert Expresion1;
assert Expresión1:Expresión2;
```

En cualquiera de los dos casos `Expresión1` tiene que ser una expresión **`boolean`** o se producirá un error de compilación. Cuando se evalúa un aserto que sólo tenga `Expresión1`, se comprueba la veracidad de la expresión y si es verdadera se continúa la ejecución del programa, pero si es falsa, se lanza una excepción de tipo **`AssertionError`**.

Si el aserto contiene además una `Expresión2` y `Expresión1` es falsa, se evalúa `Expresion2` y se le pasa como parámetro al constructor de **`AssertionError`** (Si `Expresion1` se evalúa como cierto, la segunda expresión no se evalúa).

Compilación

Los asertos se incorporaron en Java a partir del JDK 1.4, para que sean reconocidos debe compilarse con opciones de `-source 1.4` o superior.

Activación/Desactivación

Los asertos suponen un trabajo extra para el programa, trabajo que se compensa durante el desarrollo y las pruebas; pero que son superfluos cuando el programa ha sido exhaustivamente probado. No obstante, si se tiene un programa en producción y se detecta una situación de fallo, conviene reactivar los asertos para poder afinar rápidamente dónde está el error. Por esto se puede desactivar y activar la comprobación de los asertos.

Por defecto la comprobación de los asertos esta desactivada y se proporcionan dos opciones para JVM:

- `java -enableassertions` (`-ea`), para activar la comprobación.
- `java -disableassertions` (`-da`), para desactivar la comprobación.

Si estos modificadores se escriben tal cual, se activa o desactiva la comprobación de asertos para la clase que se pretende ejecutar.

Si lo que se quiere es activar/desactivar la comprobación de los asertos en un determinado paquete o de una determinada clase:

- `java -enableassertions:saludos.Hola... HolaMundo` (Activa asertos en el paquete `saludos.Hola`, por los puntos ...)
- `java -enableassertions:Saludos.Hola HolaMundo` (Activa asertos en la clase `Saludos.Hola`, por que no lleva puntos)

Y lo mismo para disable:

- `java -disablessertions:saludos.Hola... HolaMundo`
- `java -disableassertions:saludos.Hola HolaMundo`

También se puede activar para unos y desactivar para otros:

- `java -ea:Ccamion.Rueda... -da:Camion.Freno Camion.Conducir`

En resumen, la sintaxis es la siguiente:

- `java [-enableassertions | -ea] [:<package name>"..." | :<class name>]`
- `java [-disableassertions | -da] [:<package name>"..." | :<class name>]`

En el entorno de Eclipse; la verificación de asertos se activan estableciendo el campo `Default VM arguments` a `-ea`
Hay que editar la configuración del JDK:

```
[Menu Windows] -> Preferencias -> Java -> Installed JRs -> jdk
(default) -> edit
```

Ejemplos de uso

Sin asertos, si en un programa se asume que alguna condición debe ser cierta, en alguna parte del código (el invariante) se coloca como comentario:

```
if (a == 1) {
	...
}
else if (a == 2) {
	...
	}
	else {
		//cuando a == 3
		...
	}
```

Aquí es donde se puede utilizar un **`assert`**, (y en general para cualquier invariante):

```
if (a == 1) {
	...
}
else if (a == 2) {
	...
	}
	else {
		assert a == 3;
		...
	}
```

De esta manera se sigue protegiendo el `else`. Si se llega al `else` y el *aserto* no se cumple, se genera una excepción de tipo **`AssertionError`**.

Otro candidato para los asertos es una sentencia **`switch`** que no tenga cláusula **`default`**. En este caso el aserto comprobará que nunca se entra por el **`default`**, de esta manera:

```
switch (suerte){
	case Moneda.CARA:
		...
		return;
	case Moneda.CRUZ:
		...
		return;
	case default:
		assert false;
 }
```

Se puede apreciar la eficacia de utilizar **`assert false`** en cualquier lugar del programa donde se supone no se debe entrar nunca.

Puede que un programa deba hacer una comprobación férrea de determinados valores de

variables, esto se consigue incluyendo asertos que comprueben allí donde es necesario.

Si se quiere verificar que los asertos están activos para una determinada clase, se puede configurar el siguiente código estático en su constructor:

```
static {
	boolean assertsActivado = false;
	assert assertsActivado = true;
	if (!assertsActivado)
		throw new RuntimeException("¡Los asertos están desactivados!");
}
```

El código es estático, por lo que siempre se ejecutará. Si la comprobación de asertos está desactivada no se hará caso del aserto y se generará la excepción **`RuntimeException`**, en cambio si está activada, se cambiará el valor de `assertsActivado` y no se generará la excepción.

Hay que que tener en cuenta que el compilador de java no prohíbe este tipo de asertos con efectos laterales, por lo que hay que ser cuidadoso.

Conclusión

Los asertos son un buen método para comprobar valores de variables, expresiones y para comprobar estados del programa por donde no se tiene que pasar.

Si bien es cierto que esto también se puede hacer con una serie de If's, pero no sería tan eficiente, y sobre todo no se podría desactivar cuando se tiene todo acabado. Como moraleja final, se puede consultar la referencia al informe de fallo del Ariane 5 (explotó en el aire por una mala conversión de un double de 64 bits a un integer de 16 bits) donde los asertos (comprobando la buena conversión) hubiesen detectado el error durante el desarrollo; evitado la catástrofe.

Excepciones

Durante la ejecución de un programa se pueden producir situaciones -externas o internas- capaces de detenerlo o inestabilizarlo. Las situaciones anómalas en los datos están relacionadas con la semántica y disponibilidad de esos datos en programa.

Las excepciones son la manera de manifestar que ha ocurrido algo anómalo durante la ejecución normal de un programa. Se utilizan en situaciones de error que no pueden ser resueltas por el programa.

Habitualmente las excepciones son generadas y lanzadas por las diferentes partes que componen un programa; tanto las API's del lenguaje como por la parte de aplicación del usuario.

Cuando se genera una excepción, el *Administrador de Errores y Excepciones de la JVM* (Máquina Virtual Java), **atrapa la excepción, interrumpe la tarea afectada e inicia la búsqueda de un manejador adecuado.** esta búsqueda comienza por el método donde se originó y después hacia abajo en la pila de llamadas. Una vez localizado un manejador adecuado, le pasa el control para la ejecución del código que tiene asociado.

De forma general, las excepciones en Java, al igual que en otros lenguajes avanzados, constituyen un sistema interno de comunicación en las aplicaciones que permite la señalización detección, y corrección de cualquier situación que se considere anómala. **Las excepciones, utilizadas de forma adecuada; simplifican las aplicaciones, las hace más robustas y mejora la tolerancia a fallos.**

Java implementa el estilo de C++ de las excepciones para construir código flexible. Cuando ocurre un error en el programa, el código que encuentra el error lanza una excepción. Este acto de lanzar una excepción indica al proceso actual en ejecución que ha ocurrido un error. El programador puede capturar la excepción y, cuando sea posible, recuperar el control del programa. El siguiente ejemplo de programa que es una versión ampliada de **`HolaMundo`**

```
public class HolaMundo {
   public static void main(String argv[ ]) {
       String[] saludos = { "Hola mundo!",
                            "No, creo que mejor digo",
                            "HOLA MUNDO!"
                          };
       for (int i = 0; i <= 3; i++) {
           System.out.println(saludos[i]);
       }
       System.out.println("... Y también quisiera decir adiós");
   }
}
```

Después de compilar sin errores, al ejecutar, se obtiene el siguiente resultado:

```
Hola mundo!
No, creo que mejor digo
HOLA MUNDO!
java.lang.ArrayIndexOutOfBoundsException: 3
at HolaMundo2.main(HolaMundo2.java):8
```

Como puede verse, la trascendencia de tener en cuenta las posibles excepciones, es la de asegurar en lo posible, que incluso con fallos de ejecución, los programas pueden continuar y terminar controladamente; como si no tuvieran errores en su implementación.

```
public class HolaMundo {
   public static void main(String argv[ ]) {
       String[] saludos = { "Hola mundo!",
                            "No, creo que mejor digo",
                            "HOLA MUNDO!"
                          };
       for (int i = 0; i <= 3; i++) {
           try {
                System.out.println(saludos[i]);
           }
           catch (ArrayIndexOutOfBoundsException e) { }
       }
       System.out.println("... Y también quisiera decir adiós");
   }
}
```

```
Hola mundo!
No, creo que mejor digo
HOLA MUNDO!
... Y también quisiera decir adiós
```

Aspectos de las excepciones

Java dedica cinco de sus palabras reservadas a los diferentes aspectos relacionados con el ciclo de vida las excepciones: **`try`**, **`catch`**, **`finally`**, **`throw`** y **`throws`**.

Aspecto	Palabra reservada	Descripción	Alternativa de diseño
Captura	try	• Delimita un bloque de código que es monitorizado y donde pueden ser lanzadas excepciones.	• Si un bloque de código no se quiere monitorizar directamente, se puede utilizar **`throws`** en la cabecera del método donde puede aparecer una excepción para no tener que capturarla y trasladar o aplazar la captura y el tratamiento a otro método superior.
Tratamiento	catch	• Especifica el tipo de excepción que se prevé será manejada y un bloque de código alternativo que se ejecutará como respuesta prevista cuando se produzca la excepción.	
Lanzamiento	throw throws	• El lanzamiento está asociado a la adecuada sentencia de control y se requiere la instanciación con **`new`** de una excepción del tipo adecuado.	• La captura y tratamiento, rara vez se realiza en el mismo método donde se lanza. Habitualmente se declara con **`throws`** en la cabecera del método. • Si se decide no lanzar excepciones en el método afectado, es posible tratar el problema con lógica convencional. Suele ser más compleja.
Definición	class **`NuevaException`** extends **`Exception`**	• La creación de nuevos tipos de excepción tiene las mismas condiciones que crear cualquier otra clase donde interviene la herencia.	• Utilizar tipos de excepciones más genéricos, ya existentes. El programa suele quedar menos claro.

Captura y tratamiento de excepciones: try - catch

A menudo es eficaz, práctico o necesario manejar una excepción. Se puede utilizar la palabra clave **`try`** para especificar **un bloque de código monitorizado frente a todas las excepciones**. A continuación inmediatamente del bloque **`try`**, se incluye la cláusula **`catch`** que especifica **el tipo de excepción que se desea tratar**. Se pueden ver estas construcciones en el ejemplo:

```
class Excepcion {
   public static void main(String args[])  {
       try  {
```

```
            int divisor = 0;
            int dividendo = 42;
            int cociente = dividendo / divisor;
        }
        catch (ArithmeticException e)  {
            System.out.println("Intentado de dividir por cero...");
        }
    }
}
```

`ArithmeticException` es una subclase especial de **`RunTimeException`**, que describe más específicamente el tipo de error que se ha producido. El ámbito de la cláusula **`catch`** está restringido a las sentencias especificadas por la sentencia **`try`** precedente.

Para manejar una excepción, se usa la palabra **`try`** que contiene el código que puede lanzarla, este código también es llamado código protegido. Para capturar y actuar cuando se lanza una excepción, se utiliza **`catch`** para especificar la excepción a capturar y el código a ejecutar si ocurre la situación anómala prevista.

La cláusula **`catch`** no aparece en cualquier sitio, sólo pueden situarse a continuación de un bloque **`try`**. Recíprocamente, un bloque **`try`** va siempre inmediatamente seguido de uno o más bloques **`catch`**.

```
try {
   // código protegido que puede lanzar excepciones
} catch (ExClase e) {
   // código que se ejecuta si ExClase ocurre
}
```

Cláusulas catch múltiples

En algunos casos, la misma secuencia de código puede activar más de una condición excepcional. Se pueden tener varias cláusulas **`catch`** . Se inspecciona cada uno de estos tipos de excepción en el orden en que están y el primero que coincida se ejecuta. Las clases de excepción más específicas se colocarán primero, dado que no se alcanzarán las excepciones derivadas si están después de una excepción bases.

Finalizar el tratamiento

A veces es necesario estar seguro de que se ejecutará un fragmento de código dado independientemente de qué excepciones se producen y capturan. Esta es una novedad introducida por Java; permitir la ejecución de una sección de código, ocurra lo que ocurra. Este código que se describe en el bloque **`finally`**; se ejecuta tras el bloque **`try`**, y después de un eventual bloque **`catch`**.

Se ejecutará el bloque **`finally`** antes del código que está después del final del bloque **`try`**

completo.

Esto cubre pues los casos siguientes:

- El bloque **`try`** se ejecuta normalmente sin que se lance ninguna excepción
- El bloque **`try`** se ejecuta y lanza una excepción atrapada en un bloque **`catch`**
- El bloque **`try`** se ejecuta y lanza una excepción que no es atrapada en ninguno de los bloques **`catch`** que le siguen.

El interés de este bloque **`finally`** es doble. En primer lugar, permite reunir en un solo bloque un conjunto de instrucciones que de otro modo serían duplicadas:

```
try {
   //abrir un fichero
   //efectuar tratamientos susceptibles
   //de lanzar una excepción cerrar el fichero
}
catch (ExCiertaExcepcion e) {
   //tratar la excepción
   //cerrar el fichero
}
catch (ExOtroTipoExcepcion o) {
   //tratar la excepción
   //cerrar el fichero
}
```

Se tiene, gracias al bloque **`finally`**, la sintaxis siguiente:

```
try  {
   //abrir un fichero
   //efectuar tratamientos susceptibles de lanzar una excepción
}
catch (ExCiertaExcepcion e) {
   //tratar la excepción
}
catch (ExOtroTipoExcepción o) {
   //tratar la excepción
}
finally {
   //cerrar el fichero
}
```

El bloque **`finally`** permite también efectuar tratamientos tras el bloque **`try`**, aunque se haya lanzado una excepción y no haya sido atrapada en los bloques **`catch`** que siguen al bloque **`try`**:

```
try  {
   //abrir el fichero
```

```
   //tratamientos que pueden lanzar excepciones ExClase  y ExOtraClase
} catch (ExClase e) {
   //tratar la excepción
}
finally {
   //cerrar el fichero
}
```

Las instrucciones del bloque **`finally`** se ejecutarán, aunque se lance una excepción **`ExOtra`**.

La palabra **`finally`** sirve para definir el bloque de código que se ejecutará siempre, sin importar si la excepción se captura o no.

```
try {
   ...
}
catch (ExClase e) {
   ...
}
finally {
   // este bloque se ejecuta aunque no se atrape la excepción
}
```

En el siguiente ejemplo se proporciona otra versión de **`HolaMundo`**, pero ahora se captura la excepción y el programa no genera mensajes de error incontrolados:

```
public class HolaMundo2 {
   public static void main(String [ ] arg) {
       String[] saludos = { "Hola mundo!",
                                "No, creo que mejor digo",
                                "HOLA MUNDO!"
                          };
       for (int i = 0; i <= 3; i++) {
           try {
               System.out.println(saludos[i]);    // código protegido
           } catch (ArrayIndexOutOfBoundsException e) {
               System.out.println("Ha ocurrido la excepción: "
                                + e.toString());
           }
           finally {
               // siempre se ejecuta
               System.out.println("... Y también quisiera decir adiós");
           }
       }
   }
}
```

El flujo de ejecución salta al bloque **`catch`** si y solamente si se ha lanzado una excepción de tipo **`ArrayOutOfBoundsException`** o derivada de ésta en el bloque **`try`**.

Lanzamiento de excepciones

La sentencia **`throw`** se utiliza para lanzar explícitamente una excepción. En primer lugar, debe obtener un descriptor de una instancia de **`Throwable`**, mediante un parámetro en una cláusula **`catch`**, o cerrar una utilizando el operador **`new`**. Esta es la forma general de una sentencia **`throw`**:

```
throw InstanciaThrowable;
```

El flujo de la ejecución se detiene inmediatamente después de la sentencia **`throw`**, y no se llega a la sentencia siguiente. Se inspecciona el bloque **`try`** que la engloba más cercano para ver si tiene una cláusula **`catch`** cuyo tipo coincida con el de la instancia **`Throwable`**. Si la encuentra, el control se transfiere a esa sentencia. Si no, se inspecciona el siguiente bloque **`try`** que la engloba, y así sucesivamente, hasta que el gestor de excepción más externo detiene el programa e imprime el trazado de la pila hasta la sentencia **`throw`**.
Las excepciones pueden ser lanzadas por:

- El sistema.
- Cualquier método de cualquier clase que se utilice.
- El programador, en cualquier método que esté definiendo.

El mecanismo es simple:

```
//...
if (numKilometros < 0) {
   throw new ExNumKilometrosNegativo( );  // lanza la excepción
}
// las instrucciones que siguen no se ejecutan si la excepción ha sido
lanzada
//...
//...
```

El flujo de ejecución del programa se interrumpe y salta directamente al punto del programa donde se trata la excepción (**`catch`**).

Para fomentar la programación de código robusto, Java requiere que si un método hace algo que puede resultar en una excepción, entonces debe quedar claro qué acción se debe tomar si el problema se presenta. Hay dos posibilidades para cumplir este requisito:

- Usar el bloque **`try-catch`** donde se utilice una subclase de **`Exception`**, aunque el bloque **`catch`** quede vacío.
- Indicar que las excepciones no son manejadas en este método, y que por lo tanto serán aplazadas y pasadas al método que hace la llamada. Esto se hace declarando

en la cabecera del método la lista de todas las excepciones que pueden ser lanzadas (y no son tratadas) en su interior.

Declaración de excepciones y excepciones no capturadas

Muchas de las excepciones que se pueden producir en un programa las crea automáticamente Java como respuesta a alguna condición excepcional. Por ejemplo: *una división por cero*. Cuando Java intenta ejecutar la división, observa que el denominador es cero, entonces genera una excepción para que se detenga este código y se trate esta condición de error. Una vez detenido el flujo del código en la operación de división, se buscará en la pila de llamadas actual cualquier *manejador de excepciones*.

Un *manejador de excepciones* es un elemento del programa establecido para tratar inmediatamente la condición excepcional. Si no se codifica un *manejador de excepciones*, adecuado en un programa, se llegará al manejador en tiempo de ejecución por defecto de la máquina virtual Java. El manejador por defecto imprime el valor **`String`** de la excepción y el trazado de la pila desde el punto donde se produjo la excepción.

Uno de los aspectos fuertes del modelo de excepciones en Java es la obligación de la declaración. Es decir: un método debe incluir en su cabecera el conjunto de excepciones cuyo lanzamiento puede producirse en su interior y que no se tratan. El conjunto de excepciones cuyo lanzamiento puede producirse en su interior está formado por:

- Excepciones que son lanzadas en el método.
- Excepciones que son lanzadas en los métodos llamados por el método actual.

Esta declaración se realiza con el empleo de la palabra clave **`throws`** tras el nombre del método.

Ejemplo de clase, declaración y lanzamiento de excepciones:

```
//ExCantidad.java
//
public class ExCantidad extends Exception {
   private String mensaje;
   private int cantidad;

   public ExCantidad (String mensaje, int cantidad) {
       this.mensaje = mensaje;
       this.cantidad = cantidad;
   }

   public String toString() {
       return mensajes + " " + cantidad;
   }
}
```

```
//PruebaExCantidad.java
//
public class PruebaExCantidad {
   public static void main(String [ ] arg) {
       PruebaExCantidad prueba = new PruebaExCantidad ();
       for (int i = 0; i < 15; i++) {
           try  {
               prueba.muestraNum(i);
           } catch (ExCantidad e) {
               System.out.println(e.toString());
           }
       }
   }

   public PruebaExCantidad () {
       System.out.println("Ejemplo de lanzamiento de una exception");
   }

   public void muestraNum(int n) throws ExCantidad {
       if (n > 10) {
           throw new ExCantidad("Imposible mostrar el número", n);
       } else {
           System.out.println("x = " + n);
       }
   }
}
```

La declaración obligatoria de excepciones aumenta la calidad del código de dos formas complementarias.

- En primer lugar, quien escribe el método se ve obligado a declarar a continuación de la signatura, todas las excepciones susceptibles de ser lanzadas por el método. Esto obliga a ser consciente de todas las excepciones lanzadas -y no tratadas- por los métodos que son llamados en profundidad. Debe elegir, para cada una de estas excepciones, entre aplazarlas o tratarlas. No se pueden ignorarlas.
- En segundo lugar, quien utiliza el método sabe, gracias a las cláusulas **`throws`**, cuáles son las excepciones susceptibles de ser lanzadas por este método y por los métodos llamados internamente. Se puede diseñar la estrategia en función de este conjunto de excepciones, y no sólo en función de las excepciones lanzadas por el propio método.

Este esfuerzo de documentación de las excepciones tiene dos ventajas:

- Motiva para el tratamiento de los errores lo más cerca posible de su origen. Se evita la idea: *al final habrá aquí un manejador de alto nivel.*
- Aumenta la legibilidad del código y permite diseñar una estrategia real de tratamiento de excepciones a nivel del proyecto.

Esta declaración sistemática sería sin embargo poco práctica si todas las excepciones se tuvieran que declarar. En efecto, más de una decena de excepciones son definidas por el gestor de Java y son susceptibles de ser lanzadas en cualquier punto del código. **La**

declaración obligatoria aplicada estrictamente tendría como consecuencia la presencia, al principio de cada método, de un bloque parecido a este:

```
void unMetodo () throws ArithmeticException, NullPointerException,
                    IncompatibleclassChangeException, ClassCastException,
                    NegativeArraysizeException, OutofMemoryException,
                    NoclassDefFoundException, IncompatibleTypeException,
                    ArraylndexoutofBoundsException,UnsatisfiedLinkException,
                    InternalException
                    /* ... no exhaustivo ... pueden ser más */  {

// ...
}
```

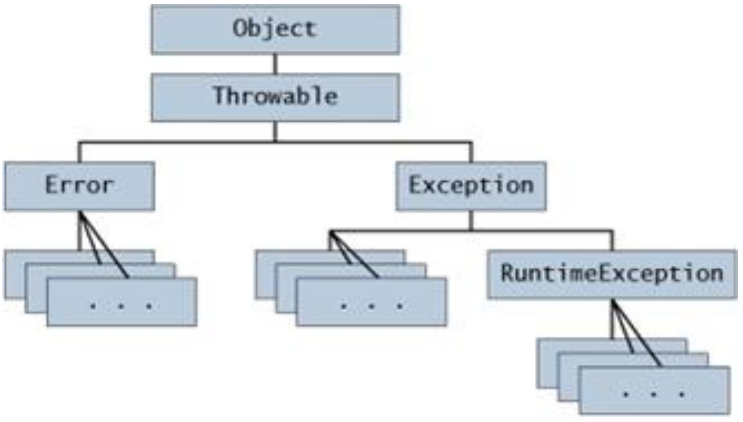
Y ¿qué ocurriría si saliera una nueva versión de Java con una excepción de sistema suplementaria, del tipo: **`ModemNotRespondingException`** Habría que reescribir todas las cabeceras de los métodos...

En el hecho descrito anteriormente se dan dos intereses contrapuestos:

- Se quiere que en los métodos se declaren todas las excepciones del interior del método no tratadas por éste.
- Se aprecia un problema engorroso al tener que declarar sistemáticamente todas las excepciones de base del lenguaje.

La solución es simple y elegante, y se basa en la utilización de la herencia y el polimorfismo. Consiste en establecer dos grandes ramas que la jerarquía de clases; en una de esas ramas es opcional declararlas: En concreto todas las **`RuntimeException`**.

Tipos de excepción



Error

Las excepciones de **`Error`**, reservadas a los errores de hardware. Definen lo que es considerado como una *condición de error grave que el programador no debería intentar recuperar*. Se pueden tratar pero en la mayoría de los casos, es aconsejable dejar que el

programa termine cuando se encuentra un error. Son siempre verificadas.

Exception

Las **`Exception`** definen *condiciones anómalas que los programas pueden detectar en tiempo de ejecución*. En vez de dejar que el programa termine, el programador puede escribir código para manejar esas excepciones y continuar o no con la ejecución del programa. Pueden definirse, lanzarse y capturarse, pero se deberán declarar si no se tratan. Son siempre verificadas.

RuntimeException

Son las excepciones lanzadas por el núcleo ejecutable, como **`ArithmeticException`**, **`NullPointerException`**, etc., heredan de la clase **`RuntimeException`**. Por convención, el compilador sabe que las subclases de **`RuntimeException`** no tienen que declararse en la cabecera de los métodos.

Al decidir que todas las excepciones que derivan directas o indirectas de la clase **`RuntimeException`** no sean declaradas, los diseñadores del lenguaje Java han resuelto inteligentemente el problema de contraposición de intereses planteado en el lenguaje, ya visto en el apartado anterior.

Así, se evita la pesadez del código y se conserva la obligación de declarar para las otras excepciones.

Las excepciones **`RuntimeException`** son definidas por el lenguaje. Pueden ser lanzadas por el programador, aunque generalmente no tiene interés, deben ser capturadas y tratadas en los programas.

Las **`RuntimeException`** deben tratarse y eventualmente lanzarlas pero, no es conveniente definir otras nuevas. No son verificadas.

Definir nuevas excepciones

Las excepciones son objetos. Esto significa que se definen en clases y que tienen todas las propiedades de los objetos.

La clase base de todas las excepciones es la clase **`Throwable`**. Sólo los objetos definidos por una subclase de **`Throwable`** pueden ser lanzados, y después eventualmente atrapados. Esta clase define un cierto número de métodos que se pueden redefinir o no. Estos métodos permiten construir y recuperar un mensaje asociado al objeto **`Exception`** afectado.

Se pueden añadir atributos y métodos que sean propios de su tipo de excepción. Así, si lanza la excepción **`ExEstacionNoEncontrada`**, puede definir como atributo de la excepción el nombre de la estación. Esto es imposible con el mecanismo de los códigos de retorno. En la

utilización de este mecanismo nos vemos obligados a almacenar en algún sitio el nombre de esta estación y recuperarlo seguidamente.

¿Cómo definir una jerarquía propia de excepciones?

El lenguaje Java no aporta una respuesta a esta cuestión conceptual, pero podemos seguir los criterios siguientes:

- Hay que distinguir las excepciones técnicas de las excepciones funcionales y no utilizarlas para gestionar errores que puedan quedar completamente depurados en la fase de desarrollo de los proyectos.
- No hay que abusar con el número de excepciones propias.
- Hay que reutilizar las excepciones existentes: las del gestor y las de los paquetes que se empleen.

Excepciones habituales

ArithmeticException

Típicamente el resultado de dividir entre cero.

```
int num = 12 / 0;
```

NullPointerException

Intento de acceder a un objeto o método antes de que sea creado o instanciado.

```
String mensaje = null;
System.out.println(mensaje);
```

NegativeArraySizeException

Intento de crear un array con un tamaño negativo.

```
String[] mensajes = new String[-3];
```

ArrayIndexOutOfBoundsException

Intento de acceder a un elemento de un array que cae fuera de los límites definidos.

```
String[] mensajes = new String[3];
mensajes[4] = "Hola";
```

SecurityException

Típicamente lanzada en un navegador, la clase **`SecurityManager`** lanza una excepción en los *applets* que intentan:

- Acceso a un archivo local.
- Abrir un socket en un host diferente al que pertenece el applet.

Ejercicios

1. (*) Comprueba si es correcto este fragmento de código ¿Por qué?

```
try {
}
finally {
}
```

2. (*) Modifica el siguiente método para que pueda compilar.

```
public static void cat(File file) {
	RandomAccessFile input = null;
	String line = null;
	try {
		input = new RandomAccessFile(file, "r");
		while ((line = input.readLine()) != null) {
			System.out.println(line);
		}
		return
	}
	finally {
		if (input != null) {
			input.close();
		}
	}
}
```

3. (**) Escribe dos versiones de un método que pide un número entero positivo por consola y muestra la raíz cuadrada de ese número. Si el número es negativo debe mostrar el mensaje *`Número no válido`*. Al terminar siempre mostrará *`Good Bye`*.
Las dos versiones serán:

a) Una que utilice lógica convencional con estructuras de control de selección.
b) Otra que utilice una **`RuntimeException`** adecuada para hacer lo indicado sin sentencias de selección.

4. (**) Escribe un método llamado `ReadNumber(int inicio, int fin)` que pide un número entero por consola en el rango `[inicio ... fin]`. En caso de que el entero de entrada no esté en el rango válido se debe lanzar una excepción adecuada. Usando este método, completa un programa que lea diez números enteros n_1, n_2, ..., n_{10} de manera que se cumpla: $1 < n_1 < \ldots < n_{10} < 100$.

5. (**) Escribe una clase llamada **`Nif`** que tiene un atributo de tipo **`String`** que encapsula el texto del *número de identificación fiscal.* Dispondrá de:

- Constructores típicos.

- ❖ Métodos de acceso.
- ❖ Redefinición de `toString()`.
- ❖ Método interno para validar la letra del nif.

En todas las operaciones que implican modificación del valor del atributo interno, la clase debe verificar:

- ❖ Que no se asigna valor **null**
- ❖ Que el formato es de ocho dígitos y una letra mayúscula al final.
- ❖ Que la letra de verificación es correcta para los dígitos indicados.

Si las verificaciones indicadas fallan, los métodos de la clase **`Nif`** lanzarán una excepción definida por el programador de la clase **`NifException`** con un mensaje informativo sobre la anomalía producida.

6. (***) Escribe una clase llamada **`DireccionPostal`** que tiene los siguientes atributos de tipo **`String`** :

- ❑ `codigoPostal`
- ❑ `via`
- ❑ `numero`
- ❑ `poblacion`

además dispondrá de:

- ❖ Constructores típicos.
- ❖ Métodos de acceso .
- ❖ Redefinición de `toString()`.
- ❖ Método interno para validar el código postal.

En todas las operaciones que implican modificación de los valor de los atributo internos, la clase debe verificar:

- ❖ Que no se asignan valores **`null`.**
- ❖ Que el formato del código postal es de cinco dígitos.
- ❖ Que el código postal existe y corresponde a la población indicada.

Si las verificaciones indicadas fallan, los métodos de la clase **`DireccioPostal`** lanzarán una excepción definida por el programador de la clase **`DirecionPostalException`** con un mensaje informativo sobre la anomalía producida.

7. (***) Escribe una clase llamada **`CorreoE`** que tiene un atributo de tipo **`String`** que encapsula el texto de una *dirección de correo electrónico.* Dispondrá de:

- ❖ Constructores típicos.
- ❖ Métodos de acceso .
- ❖ Redefinición de `toString()`.
- ❖ Método interno para validar el formato.
- ❖ Método interno para verificar que la cuenta de correo existe.

En todas las operaciones que implican modificación del valor del atributo interno, la clase debe verificar:

- ❖ Que no se asigna valor **`null`**
- ❖ Que el formato cumple la forma estándar para una dirección de correo electrónico.
- ❖ Que la dirección de correo electrónico está activa en un servidor de correo.

Si las verificaciones indicadas fallan, los métodos de la clase **`CorreoE`** lanzarán una excepción definida por el programador de la clase **`CorreoException`** con un mensaje

informativo sobre la anomalía producida.

8. (***) Escribe una clase llamada **`Usuario`** que tiene los atributos:

- ❏ `IdUsr`
- ❏ `Nombre`
- ❏ `Apellidos`
- ❏ `Nif`

Dispondrá de:

- ❖ Constructores típicos.
- ❖ Métodos de acceso .
- ❖ Redefinición de `toString()`.
- ❖ Método interno llamado `generarIdUsr()` para generar automáticamente el idUsr concatenando:
  - ➢ Tres letras mayúsculas correspondientes a las iniciales del nombre y los apellidos.
  - ➢ Los tres últimos caracteres del Nif.

  El método `generarIdUsr()` usará `validarIdUsr()` para asegurar la validez del idUsr generado. Debe capturar excepciones **`UsuarioException`** y tratarlas según las siguientes condiciones:
  - ➢ Si el `idUsr` está ya en uso se debe generar una variación cambiando la última letra por la siguiente en el alfabeto suponiendolo circular.
  - ➢ Se debe volver a validar con el método `validarIdUsr()`. Este proceso se repetirá indefinidamente.
  - ➢ Si se captura una **`UsuarioException`** asociada a un fallo de formato se debe volver a lanzar la misma excepción para que sea resuelto el problema fuera del método `generarIdUsr()`.
- ❖ Método interno llamado `validarIdUsr()` que verifica:
  - ➢ El formato correcto del `idUsr` recibido. Lanza una **`UsuarioException`** con un mensaje informativo si no se cumple el formato.
  - ➢ La no repetición del `idUsr` recibido. Lanza una **`UsuarioException`** con un mensaje informativo acorde, si ya está en uso el `idUsr`.

9. (***) Escribe un método que reciba como parámetro el nombre de un archivo de texto, lee el archivo y devuelve su contenido como texto. ¿Cuál debe ser la lógica de lanzamiento de la excepción?
Lee el capítulo que trata sobre entrada/salida y los archivos de texto.

10. (***) Escribe un método que reciba como parámetro el nombre de un archivo binario, lee el contenido del archivo y la devuelve como una matriz de bytes. Escribe un método que escribe el contenido del archivo a otro archivo. Comparar dos archivos.
Para leer el fichero:

- ❖ Utilizar **`FileInputStream`** y leer de **`ByteArrayOutputStream`**.
- ❖ Ten cuidado con el método de lectura de `read(byte[] buffer, int offset, int count)`puede que lea menos bytes que los requeridos. ¿Cuántos bytes son leídos del flujo de entrada?
- ❖ Utiliza **`try-finally`** , para cerrar los *stream*.

Fuentes y bibliografía

- MAÑAS, J. A. *Laboratorio de programación. Material de estudio. Apuntes.* [en línea] http://www.lab.dit.upm.es/~lprg/material/apuntes/index.html
- Rodríguez Galdo, Alberto. *Introducción a los Asertos en Java*. [en línea] http://www.webtaller.com/construccion/lenguajes/java/lecciones/introduccion-asertos-java-3.php
- Jean-Marc Jézéquel, IRISA y Bertrand Meyer,*Put it in the contract: The lessons of Ariane*. [en línea] http://www.irisa.fr/pampa/EPEE/Ariane5.html
- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en línea][traducción automática del búlgaro] http://www.introprogramming.info/intro-java-book/
- SÁNCHEZ, J. Java2. [en línea] http://www.jorgesanchez.net/programacion/manuales/Java.pdf
- Oracle. *The Java Tutorial.* [en línea] http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html
- Oracle. *Java™ Platform, Standard Edition 8* API Specification. [en línea] http://docs.oracle.com/javase/8/docs/api/