4. TRATAMIENTO DE EXCEPCIONES.

Cuando un programa Java viola las restricciones semánticas del lenguaje (se produce un error), la máquina virtual Java comunica este hecho al programa mediante una **excepción**.

Muchas clases de errores pueden provocar una excepción, desde un desbordamiento de memoria o un disco duro estropeado hasta un disquete protegido contra escritura, un intento de dividir por cero o intentar acceder a un vector fuera de sus límites. Cuando esto ocurre, la máquina virtual Java crea un objeto de la clase `exception` o `error` y se notifica el hecho al sistema de ejecución. Se dice que se ha lanzado una excepción ("*Throwing Exception*").

Un método se dice que es capaz de tratar una excepción ("*Catch Exception*") si ha previsto el error que se ha producido y prevé también las operaciones a realizar para "recuperar" el programa de ese estado de error.

En el momento en que es lanzada una excepción, la máquina virtual Java recorre la pila de llamadas de métodos en busca de alguno que sea capaz de tratar la clase de excepción lanzada. Para ello, comienza examinando el método donde se ha producido la excepción; si este método no es capaz de tratarla, examina el método desde el que se realizó la llamada al método donde se produjo la excepción y así sucesivamente hasta llegar al último de ellos. En caso de que ninguno de los métodos de la pila sea capaz de tratar la excepción, la máquina virtual Java muestra un mensaje de error y el programa termina.

Los programas escritos en Java también pueden lanzar excepciones explícitamente mediante la instrucción `throw`, lo que facilita la devolución de un "código de error" al método que invocó el método que causó el error.

Como primer encuentro con las excepciones, pruebe a ejecutar el siguiente programa:

```
class Excepcion {
      public static void main(String argumentos[]) {
          int i=5, j=0;
          int k=i/j; // División por cero
      }
}
```

Produce la siguiente salida al ser ejecutado:

```
java.lang.ArithmeticException: / by zero
        at Excepcion.main(Excepcion.java:4)
```

Lo que ha ocurrido es que la máquina virtual Java ha detectado una condición de error y ha creado un objeto de la clase `java.lang.ArithmeticException`. Como el método donde se ha producido la excepción no es capaz de tratarla, se trata por la máquina virtual Java, que muestra el mensaje de error anterior y finaliza la ejecución del programa.

4.1 Lanzamiento de excepciones (throw).

Como se ha comentado anteriormente, un método también es capaz de lanzar excepciones. Por ejemplo, en el siguiente programa se genera una condición de error si el dividendo es menor que el divisor:

```
class LanzaExcepcion {
      public static void main(String argumentos[])
                  throws ArithmeticException {
         int i=1, j=2;
         if (i/j< 1)
            throw new ArithmeticException();
         else
            System.out.println(i/j);
      }
}
```

Genera el siguiente mensaje:

```
java.lang.ArithmeticException
        at
LanzaExcepcion.main(LanzaExcepcion.java:5)
```

En primer lugar, es necesario declarar todas las posibles excepciones que es posible generar en el método, utilizando la cláusula `throws` de la declaración de métodos[35].

Para lanzar la excepción es necesario crear un objeto de tipo `Exception` o alguna de sus subclase (por ejemplo: `ArithmeticException`) y lanzarlo mediante la instrucción `throw`.

4.2 Tratamiento de excepciones.

En Java, de forma similar a C++ se pueden tratar las excepciones previstas por el programador utilizando unos mecanismos, los manejadores de excepciones, que se estructuran en tres bloques:

- El bloque `try`.
- El bloque `catch`.
- El bloque `finally` (no existente en C++).

4.2.1 Manejador de excepciones.

Un manejador de excepciones es una porción de código que se va a encargar de tratar las posibles excepciones que se puedan generar.

[35] Ver punto **3.4.1.7** Lista de excepciones potenciales. En página 101.

4.2.1.1 El bloque try.

Lo primero que hay que hacer para que un método sea capaz de tratar una excepción generada por la máquina virtual Java o por el propio programa mediante una instrucción `throw` es encerrar las instrucciones susceptibles de generarla en un bloque `try`.

```
try {
    BloqueDeIntrucciones
}
```

Cualquier excepción que se produzca dentro del bloque try será analizado por el bloque o bloques `catch` que se verá en el punto siguiente. En el momento en que se produzca la excepción, se abandona el bloque `try` y, por lo tanto, las instrucciones que sigan al punto donde se produjo la excepción no serán ejecutadas.

Cada bloque `try` debe tener asociado por lo menos un bloque `catch`.

4.2.1.2 El bloque catch.

```
try {
    BloqueDeIntrucciones
} catch (TipoExcepción nombreVariable) {
        BloqueCatch
} catch (TipoExcepción nombreVariable) {
        BloqueCatch
}
```

Por cada bloque `try` pueden declararse uno o varios bloques `catch`, cada uno de ellos capaz de tratar un tipo u otro de excepción.

Para declarar el tipo de excepción que es capaz de tratar un bloque `catch`, se declara un objeto cuya clase es la clase de la excepción que se desea tratar o una de sus superclases.

```
class ExcepcionTratada {
      public static void main(String argumentos[]) {
          int i=5, j=0;
          try {
              int k=i/j;
              System.out.println("Esto no se va a ejecutar.");
          } catch (ArithmeticException ex) {
            System.out.println("Ha intentado dividir por cero");
            }
          System.out.println("Fin del programa");
      }
}
```

Salida:

```
Ha intentado dividir por cero
Fin del programa
```

Cuando se intenta dividir por cero, la máquina virtual Java genera un objeto de la clase `ArithmeticException`. Al producirse la excepción dentro de un bloque `try`, la ejecución del programa se pasa al primer bloque `catch`. Si la clase de la excepción `se` corresponde con la clase o alguna subclase de la clase declarada en el bloque `catch`, se ejecuta el bloque de instrucciones `catch` y a continuación se pasa el control del programa a la primera instrucción a partir de los bloques `try-catch`.

Como puede verse, también se podría haber utilizado en la declaración del bloque `catch`, una superclase de la clase `ArithmeticException`. Por ejemplo:

```
catch (RuntimeException ex)
```

o

```
catch (Exception ex)
```

Sin embargo, es mejor utilizar excepciones lo más cercanas al tipo de error previsto, ya que lo que se pretende es recuperar al programa de alguna condición de error y si "se meten todas las condiciones en el mismo saco", seguramente habrá que averiguar después qué condición de error se produjo para poder dar una respuesta adecuada.

4.2.1.3 El bloque finally.

```
try {
    BloqueDeIntrucciones
} catch (TipoExcepción nombreVariable) {
        BloqueCatch
} catch (TipoExcepción nombreVariable) {
        BloqueCatch
}
} catch (TipoExcepción nombreVariable) {
        BloqueCatch
}
finally {
        BloqueFinally
}
```

El bloque `finally` se utiliza para ejecutar un bloque de instrucciones sea cual sea la excepción que se produzca. Este bloque se ejecutará en cualquier caso, incluso si no se produce ninguna excepción. Sirve para no tener que repetir código en el bloque `try` y en los bloques `catch`.

4.3 Jerarquía de excepciones.

Las excepciones son objetos pertenecientes a la clase `Throwable` o alguna de sus subclases.

Dependiendo del lugar donde se produzcan existen dos tipos de excepciones:

1) Las excepciones **síncronas** no son lanzadas en un punto arbitrario del programa sino que, en cierta forma, son previsibles en determinados puntos del programa como resultado de evaluar ciertas expresiones o la invocación de determinadas instrucciones o métodos.

2) Las excepciones **asíncronas** pueden producirse en cualquier parte del programa y no son tan "previsibles". Pueden producirse excepciones asíncronas debido a dos razones:

- La invocación del método `stop()` de la clase `Thread` que se está ejecutando. (Véase el capítulo 6 **Threads.** en la página 199).

- Un error interno en la máquina virtual Java.

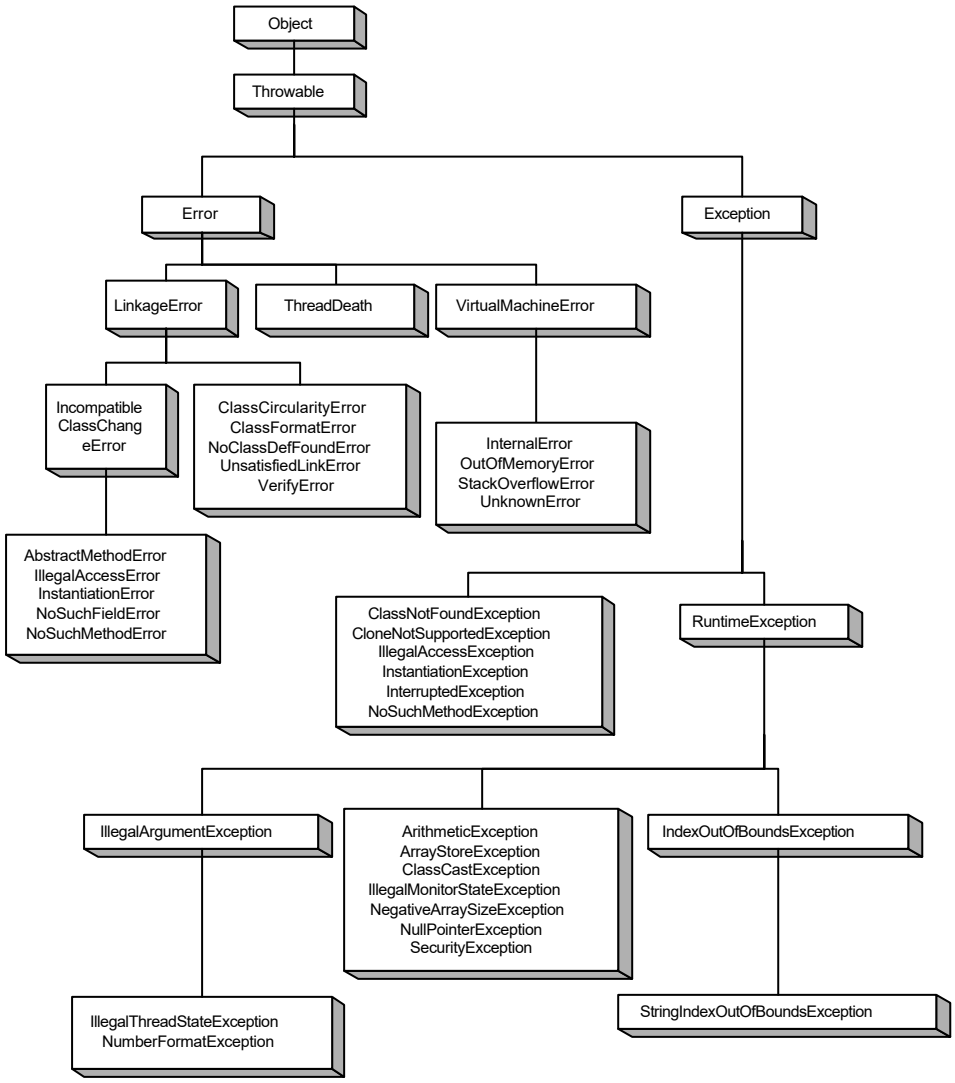
Dependiendo de si el compilador comprueba o no que se declare un manejador para tratar las excepciones, se pueden dividir en:

1) Las excepciones **comprobables** son repasadas por el compilador Java durante el proceso de compilación, de forma que si no existe un manejador que las trate, generará un mensaje de error.

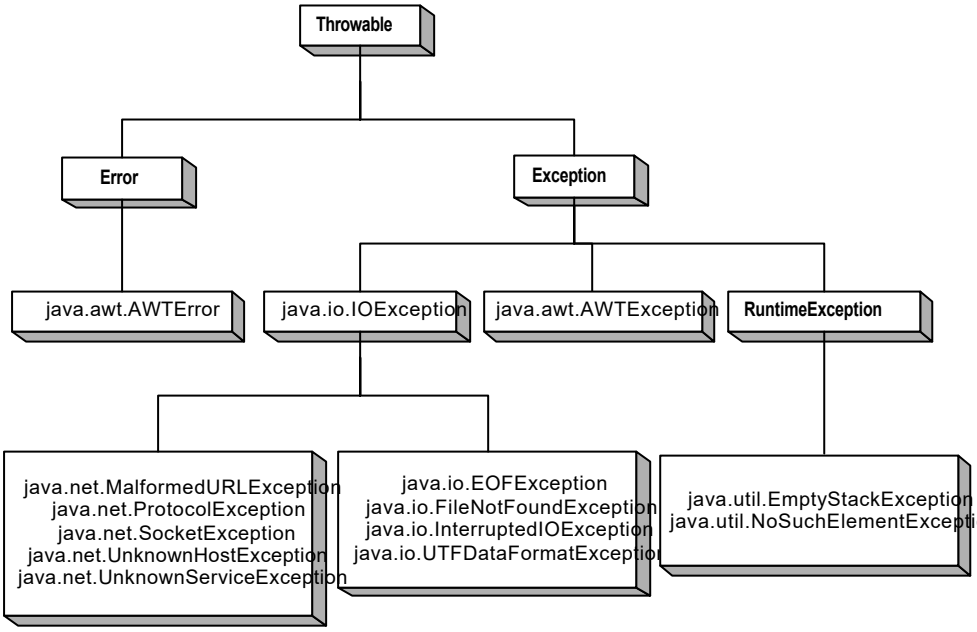
2) Las excepciones **no comprobables** son la clase `RuntimeException` y sus subclases junto con la clase `Error` y sus subclases.

También pueden definirse por el programador subclases de las excepciones anteriores. Las más interesantes desde el punto de vista del programador son las subclases de la superclase **`Exception`** ya que éstas pueden ser comprobadas por el compilador.

La jerarquía de excepciones existentes en el paquete `java.lang` es la siguiente:



Las excepciones en otros paquetes Java son:



```
class Nif {
      int dni;
      char letra;
      static final char tabla[]={'T','R','W','A','G','M','Y',
                                 'F','P','D','X','B','N','J',
                          'Z','S','Q','V','H','L','C','K','E'};

      public Nif(int ndni,char nletra) throws NifException{
      // debido a la cláusula throws, este método es capaz
      // de generar excepciones de la clase NifException
          if (Character.toUpperCase(nletra)==tabla[ndni%23]) {
             // si la letra es correcta, almacenarla en el atributo
             dni=ndni;
             letra=Character.toUpperCase(nletra);
             }
          else
            // si la letra es incorrecta, generar una excepción
            throw new LetraNifException("Letra de NIF incorrecta");
```

```
}

public Nif(int ndni) {
    dni=ndni;
    letra=tabla[dni%23];
}

public Nif(String sNif) throws NifException,
                               LetraNifException {
// debido a la cláusula throws, este método es capaz
// de generar excepciones de la clase NifException y
// de la clase LetraNifException, aunque en realidad
// sería suficiente con NifException, ya que la clase
// LetraNifException es una subclase de ésta.
    char letraAux;
    StringBuffer sNumeros= new StringBuffer();
    int i,ndni;
    for (i=0;i<sNif.length();i++) {
        if ("1234567890".indexOf(sNif.charAt(i))!=-1) {
           sNumeros.append(sNif.charAt(i));
        }
    }
    try {
        dni=Integer.parseInt(sNumeros.toString());
        letraAux=Character.toUpperCase(sNif.charAt(
                                        sNif.length()-1));
    } catch (Exception ex) {
      // este bloque catch intercepta cualquier tipo de
      // excepción, incluso NumberFormatException
        throw new NifException("NIF incorrecto");
      }
    letra=tabla[dni%23];
    if ("ABCDEFGHIJKLMNOPQRSTUVWXYZ".indexOf(letraAux)!=-1) {
       // es una letra correcta
       if (letraAux!=letra) {
          // pero no la adecuada para el NIF
          throw new
                LetraNifException("Letra de NIF incorrecta");
       }
    } else letra=tabla[dni%23];
}

public char obtenerLetra() {
    return letra;
}

public int obtenerDni() {
    return dni;
}

public String toString() {
// redefinición del método toString() para que devuelva
```

```
        // un valor más significativo.
            return (String.valueOf(dni)+String.valueOf(letra));
        }

        public String toStringConFormato() {
            String sAux= String.valueOf(dni);
            StringBuffer s = new StringBuffer();
            int i;
            for (i=sAux.length()-1;i>2;i-=3) {
                s.insert(0,sAux.substring(i-2,i+1));
                s.insert(0,".");
                }
            s.insert(0,sAux.substring(0,i+1));
            s.append('-');
            s.append(letra);
            return (s.toString());
        }

        static char letraNif(int ndni) {
            return tabla[ndni%23];
        }

        static char letraNif(String sDni) throws NifException {
            Nif j = new Nif(sDni);
            return j.obtenerLetra();
        }
}

class NifException extends Exception {
        public NifException() { super(); }
        public NifException(String s) { super(s); }
}

class LetraNifException extends NifException {
        public LetraNifException() { super(); }
        public LetraNifException(String s) { super(s); }
}
```

En este programa se declara una clase **Nif** con dos atributos: dni, que almacena el número de DNI; y letra, que almacena la letra del NIF.

Se han declarado dos tipos de excepciones:

- La clase NifException como subclase de la clase `Exception`.
- La clase LetraNifException como subclase de la anterior.

Las excepciones tienen dos constructores, uno sin parámetros y otro que acepta un `String` (correspondiente al mensaje de error que se mostrará por la máquina

virtual). En el ejemplo se han sobrescrito los constructores para introducir un mensaje de error en la excepción (“Letra de NIF incorrecta” y “NIF incorrecto” respectivamente). Si no se desea introducir ningún mensaje, no es necesario sobrescribir los constructores y la declaración de las Excepciones es más simple:

```
class NifException extends Exception {
}
class LetraNifException extends NifException {
}
```

Por ser subclases de la superclase `Exception`, ambos tipos de excepción son comprobables en tiempo de compilación.

El siguiente programa generaría errores de compilación:

```
class VerificaNif {
      public static void main(String argumentos[]) {
          Nif n;
          if (argumentos.length!=1) {
             System.out.println("Uso: VerificaNif NIF");
             return;
          }
          else {
             n = new Nif(argumentos[0]);
             System.out.println("Nif: "+n.toStringConFormato());
          }
      }
}
```

Los errores de compilación generados son:

```
VerificaNif.java:9: Exception LetraNifException must
be caught, or it must be declared in the throws clause
of this method.
            n = new Nif(argumentos[0]);
                ^
VerificaNif.java:9: Exception NifException must be
caught, or it must be declared in the throws clause of
this method.
            n = new Nif(argumentos[0]);
                ^
2 errors
```

NOTA: Si la clase `NifException` se hubiera declarado como subclase de `RuntimeException` en lugar de `Exception`, el compilador no mostraría ningún mensaje de error, ya que la clase `RuntimeException` es NO comprobable en tiempo de compilación.

Si el método `main()` se hubiera declarado de esta forma:

public static void main(String argumentos[]) throws NifException {

Entonces no se habrían generado los errores de compilación, ya que mediante esta declaración se está indicando que las excepciones de la clase (o subclases) NifException no se tratan en este método sino que serán tratadas en el método que realice una llamada al mismo (en este caso no hay ninguno, únicamente la máquina virtual Java). En el caso en que se introduzca un Nif incorrecto de la siguiente forma:

java VerificaNif 18957690R

Se creará una excepción que es tratada por la máquina virtual, que lo que hace es mostrar un mensaje y finalizar el programa:

```
LetraNifException: Letra de NIF incorrecta
        at Nif.<init>(Nif.java:32)
        at
VerificaNif.main(VerificaNif.java:9)
```

Si lo que se pretende es que la excepción sea tratada por el propio método `main()`, hay que declarar un bloque `try` junto con los bloques `catch` adecuados:

```
class VerificaNif {
      public static void main(String argumentos[]) {
          Nif n;
          if (argumentos.length!=1) {
             System.out.println("Uso: VerificaNif NIF");
             return;
          }
          else {
             try {
                 n = new Nif(argumentos[0]);
                 System.out.println("El NIF es correcto.");
                 }
                 catch (LetraNifException ex) {
                       System.out.println(
                         "La letra del NIF es incorrecta");
                 }
                 catch (NifException ex) {
                       System.out.println(
                         "Construcción de NIF incorrecta");
                 }
                 finally {
                       System.out.println("Que tenga un buen día");
                 }
          }
      }
}
```

En este ejemplo, el programa acepta un NIF en la línea de comandos. En caso de que el NIF sea correcto, no se produce ninguna excepción y se ejecuta la línea: `System.out.println("El NIF es correcto.");`

Si al crear el NIF (`new Nif (argumentos[0]);`) se produce un error, entonces esa excepción será comprobada por el primer bloque `catch`. Si la excepción se corresponde con la clase LetraNifException, se muestra el mensaje: La letra del NIF es incorrecta.

Si la excepción generada no pertenece a la clase LetraNifException, se comprueba si pertenece a la clase NifException, en cuyo caso, se mostraría el mensaje: `Construcción de NIF incorrecta.`

En el caso de que no hubiéramos querido diferenciar entre una letra de NIF incorrecta y un NIF construido incorrectamente, podríamos haber utilizado un

solo bloque `catch` con la clase NifException, ya que esta clase es superclase de LetraNifException y por lo tanto abarca a ambas excepciones.

Por último, en cualquier caso, tanto si se produce una excepción como si no, o incluso si se produce una excepción no contemplada en los bloques `catch`, se mostrará el mensaje:

```
Que tenga un buen día.
```

4.4 Ventajas del tratamiento de excepciones.

Las ventajas de un mecanismo de tratamiento de excepciones como este son varias:

Separación del código "útil" del tratamiento de errores.

- Propagación de errores a través de la pila de métodos.
- Agrupación y diferenciación de errores.
- Claridad del código y obligación del tratamiento de errores.

4.4.1 Separación del código útil del tratamiento de errores.

Supóngase que se quiere realizar un bloque de instrucciones que realiza un procesamiento secuencial de un fichero:

```
{
abrir_fichero("prueba");
Mientras ! Fin_Fichero("prueba") {
       auxiliar = leer_registro("prueba");
       procesar(auxiliar);
       }
cerrar_fichero("prueba");
}
```

Si se quiere tener en cuenta los posibles errores, debería hacerse algo parecido a esto:

```
{
coderror = abrir_fichero("prueba"));
if (coderror ==0) {
       Mientras( ! Fin_Fichero("prueba")) y ( coderror==0 ) {
              auxiliar = leer_registro("prueba");
```

```
                if (auxiliar != ERROR) {
                        coderror=procesar(auxiliar);
                        if (coderror != 0)
                                coderr = -3;
                        };
                else
                        coderror = -2;
                }
        if(cerrar_fichero("prueba")!=0)
                coderror=-4;
        };
else
        coderror = -1;
}
```

Por último faltaría tratar cada uno de los posibles errores, por ejemplo mediante una construcción del tipo switch. Nótese que las instrucciones "útiles" son las que se encuentran en **negrita**.

En Java, el código sería algo parecido a esto:

```
try {
        abrir_fichero("prueba");
        Mientras ! Fin_Fichero("prueba") {
                auxiliar = leer_registro("prueba");
                procesar(auxiliar);
                }
        cerrar_fichero("prueba");
        } catch (ExceptionAbrirFichero) {
                Hacer algo;
        } catch (ExceptionLeerRegistro) {
                Hacer algo;
        } catch (ExceptionProcesar) {
                Hacer algo;
        } catch (ExceptionCerrarFichero) {
                Hacer algo;
        }
```

En este caso se ha incluido la estructura equivalente a la que falta en el caso anterior (switch).

En Java, el código útil (en negrita) se encuentra agrupado. De esta forma se consigue una mayor claridad en el código que realmente interesa.

4.4.2 Propagación de errores a través de la pila de métodos.

Supóngase que existen cuatro métodos y que el primero de ellos es el interesado en procesar una condición de error que se produce en el cuarto de ellos:

```
método1 ( ) {
       int error;
       error = método2 ( );
       if ( error != 0 )
              Procesar Error;
       else
              Hacer algo;
}
int método2 ( ) {
       int error;
       error = método3 ( );
       if ( error != 0 )
              return error;
       else
              Hacer algo;
}
int método3 ( ) {
       int error;
       error = método4 ( );
       if ( error != 0 )
              return error;
       else
              Hacer algo;
}
int método4 ( ) {
       int error;
       error = Proceso que puede generar un error ( );
       if ( error != 0 )
              return error;
       else
              Hacer algo;
}
```

Todos y cada uno de los métodos deben tener en cuenta el posible error y adaptar su código para tratarlo, así como devolver el código de error al método que lo invocó.

En Java:

```
método1 ( ) {
       try {
              método2 ( );
              Hacer algo;
```

```
        } catch (Exception ex)
              Procesar Error;
        }
}
int método2 ( ) throws Exception{
        método3 ( );
        Hacer algo;
}
int método3 ( ) throws Exception{
        método4 ( );
        Hacer algo;
}
int método4 ( ) throws Exception{
        Proceso que puede generar un error ( );
        Hacer algo;
}
```

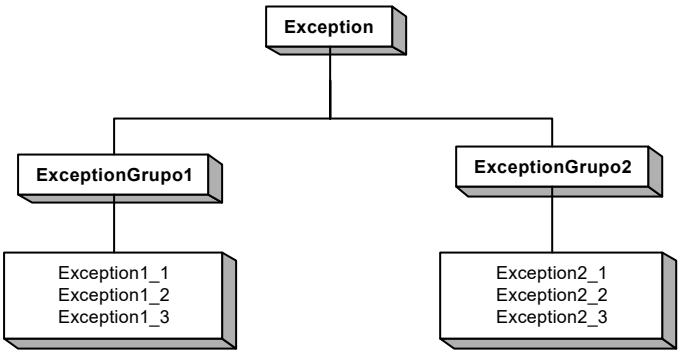
Cada uno de los métodos únicamente declara mediante la cláusula `throws` el /los error /errores que puede generar. Éstos se propagan automáticamente a través de la pila de llamadas sucesivas de métodos. El método interesado en tratar el error es el único que debe encargarse de "tratarlo". Los demás métodos únicamente deben "ser conscientes" del error que puede producirse al llamar a otro método.

Si además el error se declara como una subclase de `Exception`[36], el compilador se encarga de mostrar un mensaje de error en caso de que el método no declare la cláusula **throws** adecuada (o trate la excepción), lo cual facilita la tarea del programador.

[36] Una excepción subclase de Exception es **comprobable** en tiempo de compilación. El compilador detecta que no se ha tenido en cuenta la excepción que puede generarse al llamar a un método.

4.4.3 Agrupación y diferenciación de errores.

Si se ha declarado una jerarquía de excepciones como la siguiente:



De esta forma:

```
class ExceptionGrupo1 extends Exception { }
class ExceptionGrupo2 extends Exception { }
class Exception1_1 extends ExceptionGrupo1 { }
class Exception1_2 extends ExceptionGrupo1 { }
class Exception1_3 extends ExceptionGrupo1 { }
class Exception2_1 extends ExceptionGrupo2 { }
class Exception2_2 extends ExceptionGrupo2 { }
class Exception2_3 extends ExceptionGrupo2 { }
```

Las excepciones pueden ser tratadas particularmente:

```
try {
      …
} catch (Exception1_2 ex) {
      …
}
```

O pueden tratarse como grupos:

```
try {
      …
} catch (ExceptionGrupo1) {
      …
}
```

En este caso se detectarían las excepciones Exception1_1, Exception1_2 y Exception1_3 además de las excepciones de la clase ExceptionGrupo1.

También podrían detectarse todos los tipos de excepciones mediante: `catch (Exception exc)`.

4.4.4 Claridad del código y obligación del tratamiento de errores.

Como se ha visto en los puntos anteriores, el código resulta más claro ya que:

- Se separa el código "útil" del de tratamiento de errores.
- Se acerca el código de tratamiento de errores al método que realmente es el interesado en tratarlos.
- Cada tipo de error se trata en un bloque `catch` diferenciado.

Además, se obliga al programador a tratar los errores que puede generar un método, suponiendo que las excepciones declaradas correspondientes a los distintos tipos de error sean subclases de la clase `Exception`, que es comprobable por el compilador.

Quien programa un método declara las excepciones que puede generar mediante la cláusula `throws`**.** Si otro programador (o el mismo) desea utilizar ese método deberá utilizar una estructura `try-catch` para tratarlo o en caso contrario lanzarlo mediante la cláusula `throws` para que sea el método que invoque al mismo quien lo trate.

En cualquier caso, el programador que desea utilizar un método es consciente, porque el compilador se lo señala, de los tipos de errores (excepciones) que puede generar.

4.5 CUESTIONES

1. ¿Qué es una excepción dentro de un programa Java?

2. Explica cuál es la finalidad del bloque `try`. Pon un ejemplo de su uso.

3. ¿Cuándo se ejecuta el código que contiene un bloque `catch`?

4. ¿Qué instrucción empleamos para "lanzar" una excepción? ¿Para qué nos sirve lanzar una excepción?

5. El bloque `finally` nos permite especificar una acción a realizar después del bloque `try`, tanto si se produce una excepción como si no. ¿Qué ventajas aporta esto?

6. ¿Son todas las excepciones comprobables en tiempo de compilación? ¿Por qué?

7. Hay lenguajes que carecen de mecanismos de tratamiento de excepciones. ¿Qué ventajas crees que aporta este mecanismo en Java?

8. Cuando se produce una excepción en un método y no se ha declarado un tratamiento para la misma ¿qué sucede? ¿quién se encarga de su tratamiento?