Entornos de desarrollo
Bloque 2 Tema 4: Funciones y manejo de excepciones

por Nacho Iborra

2.4.1. Funciones

Las funciones nos permiten agrupar nuestro código de forma que podamos reutilizar un trozo del mismo tantas veces como sea necesario sin tener que duplicarlo. Simplemente se asigna un nombre a este trozo de código (nombre de la función) y podemos *llamar* a este código desde otras partes del programa. Este paradigma se llama **programación modular**, ya que define *módulos* o funciones para agrupar nuestro código en pequeñas subtateas, y llamar a cada subtarea cuando sea necesario.

2.4.1.1. Definición de una función

Para usar funciones dentro de una clase, se pueden declarar como `public static` si se pretende llamarla desde cualquier sitio. Aprenderemos más sobre como declarar otros tipos de funciones y como establecer su visibilidad, pero por ahora nos centraremos en las funciones public static.

En la definición de una función, se debe especificar el tipo de dato que devolverá (o `void`si no devuelve nada), el nombre de la función y un par de paréntesis:

```
public static String myFunction()
{
    return "Hello";
}
```

Se puede llamar a esta función desde otra función de la misma clase (incluyendo la función `main`) escribiendo el nombre de la función y los paréntesis:

```
public static void main(String[] args)
{
    String result = myFunction();    // result = "Hello"
    System.out.println(result);
}
```

Observad que las funciones siempre comienzan en minúsculas en Java (en C# comenzaban en mayúsculas si eran public).

2.4.1.2. Parámetros de una función

Algunas funciones necesitan algunos datos adicionales para realizar su tarea. Estos datos se pasan a la función como **parámetros**, una clase de variables que se especifican dentro de los paréntesis. Se escribe el tipo y el nombre de cada parámetro. Por ejemplo:

```
public static void myOtherFunction (int a, String b)
{
    ...
}

public static void main (int a, String b)
{
    ...
    myOtherFunction(3, "Hello");
}
```

2.4.1.3. Parámetros pasados por valor o por referencia

En muchos lenguajes de programación podemos escoger como pasar los parámetros, por valor o por referencia. En Java, todos los parámetros se pasan SIEMPRE por valor. O sea, si cambiamos el valor original del parámetro dentro de la función, el cambio no tendrá efecto cuando la función finalice.

```
public static void myFunction(int value)
{
    // This increment will have no effect after exiting the function
    value = value + 1;
}

public static void myOtherFunction(Integer value)
{
        // This assignment has no effect out of the function
        value = new Integer(10);
}
```

En cuanto a los **arrays**, no se podrá asignar un array completo pasado como parámetro, pero sí reasignar cada valor interno.

```
public static void myFunctionWithArray(int[] data)
{
    data[0] = 10;         // OK
    data = new int[10];  // has no effect after the function
}
```

Trabajando con objetos (no tipos primitivos), se pueden cambiar sus propiedades internas o atributos llamando a sus métodos. De nuevo, no se puede reasignar la variable a otro objeto. Por ejemplo:

```
public static void myFunctionWithObject(Person person)
{
    person = new Person(...); // Error
    person.setName("Nacho");  // Name successfully changed
}
```

2.4.1.4. Parámetros de la función *main*

La función *main* tiene un array de String como parámetro:

```
public static void main(String[] args)
{
    ...
}
```

Esto significa que se pueden pasar tantos argumentos como se necesiten a la función *main* desde la línea de comandos. El primer argumento será el correspondiente a la posición 0, el segundo la posición 1 etc.

```
public static void main(String[] args)
{
    if (args.length > 0)
        System.out.println("Received " + args.length + " args.");
}
```

> **Ejercicios propuestos**:
>
> **2.4.1.1.** Crea un programa llamado *Palindrome* con una función llamada *isPalindrome*. Esta función recibirá una cadena como parámetro y devolverá un booleano indicando si dicha cadena es palíndromo (o sea, una cadena que se lee igual de adelante hacia atrás que de atrás hacia adelante, sin tener en cuenta mayúsculas ni minúsculas). Prueba esta función desde la función *main* con los textos *Hannah*, *Too hot to hoot* and *Java is the best language* (este último no es palíndromo).
>
> **2.4.1.2** Crea un programa llamado *CountOcurrences* con una función llamada *countString*. Esta función recibirá dos cadenas *a* y *b*, y un entero *n* como parámetros, y devolverá un booleano indicando si la cadena *b* está contenida al menos *n* veces en la cadena *a*. Pruébalo desde la función main con la cadena a= `This string is just a sample string`, la subcadena b= `string` y el número n= 2 (debería devolver `true`).

2.4.2. Manejo de excepciones

Durante el desarrollo de un programa en Java se pueden encontrar dos tipos de errores: errores de compilación y errores de ejecución. Los primeros se detectan con el compilador en la creación del código, pero los errores de ejecución son difíciles de predecir en general: errores de red, divisiones por cero, fichero no encontrado... La mayoría de ellos pueden ser manejados con **excepciones**.

Una excepción es un evento que ocurre durante la ejecución de un programa y lo hace salir de su flujo normal de instrucciones. Así que una forma hábil de lidiar con estos errores será separar el código *normal* del error. Cuando una excepción ocurra, se dice que se lanza **thrown**, y se puede elegir entre propagarlo (lanzarlo otra vez) o capturarlo **catching** y procesar el error.

2.4.2.1. Errores de ejecución y tipos de excepciones

Los errores de ejecución pueden ser de dos tipos principales:

- **Errors**: en este caso se trata de errores ***fatales*** que ocurren durante la ejecución de un programa, tales como errores de hardware, errores de memoria... Estos errores no pueden ser manejados en una aplicación Java.
- **Exceptions**: Son errores no críticos que pueden ser manejados (fichero no encontrado...) Dentro de estos tipos de errores podemos hablar de:
  - **Runtime exceptions**: No necesitan ser capturados ***catch***, y son difíciles de predecir en general. Por ejemplo, la asignación a null de una variable, o salirse de los límites de un array.
  - **Checked exceptions**: Estas excepciones deben ser capturadas, o declararlas para ser lanzadas *thrown*. O sea, si se usa una función que puede generar este tipo de excepciones, el compilador se quejará si no las capturamos o las lanzamos. Por ejemplo, siempre que se llama a la instrucción `Thread.sleep` es necesario capturar o lanzar una `InterruptedException`.

Todos los tipos de excepción son un subtipo del tipo principal `Exception`. Este tipo genérico almacena el mensaje de error producido por la excepción. Hay algunos subtipos que almacenan información más específica. Por ejemplo, `ParseException` es un subtipo de `Exception`que se lanza siempre que las datos no pueden ser analizados adecuadamente. Almacena el mensaje de error junto con la posición donde el error ha sido encontrado.

2.4.2.2. Capturar excepciones

Cualquier trozo de código puede lanzar una excepción, se pueden capturar usando un bloque `try..catch`. Se pondrá dentro del `try` el código del programa que podría causar una excepción, y dentro del `catch`las instrucciones para responder al error específico. Se puede mostrar un mensaje de error, devolver un valor determinado, entre otras posibles opciones. Este ejemplo intenta convertir una cadena a entero. Si la conversión no se puede llevar a cabo porque la entrada no es un valor válido, entonces se generará una excepción del tipo `NumberFormatException` y se puede crear un error apropiado dentro de la cláusula `catch`.

```
int number;
string text = ... // Whatever value

try
{
    number = Integer.parseInt(text);
} catch (NumberFormatException e) {
    System.err.println("Error parsing text: " + e.getMessage());
}
```

El método `getMessage` recoge el mensaje de error producido por la excepción. Se puede usar también el método `printStackTrace` para la traza del error. Muestra la pila de llamadas de los distintos métodos hasta

que se produjo el error. Se pueden añadir tantos bloques `catch` como sean necesarios, y cada uno representará un tipo de excepción específico:

```
try
{
    // Code that may fail
} catch (NumberFormatException e1) {
    // Error message for number format
} catch (ArithmeticException e2) {
    // Error message for dividing by zero
...
} catch (Exception eN) {
    // Error message for any other error
}
```

Sin embargo, se deben poner estas cláusulas `catch` en orden, de forma que la más genérica se coloque al final, ya que el programa entrará en el primer `catch`que se coincida con la excepción producida. En decir, si se pone la línea `catch (Exception)` al principio, el resto de cláusulas no tendrán efecto, porque cualquiera de ellas es un subtipo de `Exception` y por tanto será capturada por la primera cláusula.

Hay muchas instrucciones que fuerzan el tratamiento de una excepción específica. Por ejemplo, como se ha dicho antes, se se llama a la instrucción `Thread.sleep`, el compilador nos exigirá que tratemos la excepción `InterruptedException`. Se puede hacer de esta forma:

```
try
{
    Thread.sleep(5000);
} catch (InterruptedException e) {
    System.err.println("Interruption during sleep: " + e.getMessage());
}
```

Sin embargo siempre se podrá usar el elemento `Exception` en el `catch` para tratar cualquier tipo de excepción.

2.4.2.3. Lanzar excepciones

La segunda forma de tratar las excepciones es lanzarlas. De esta forma, se pasan a la siguiente función en la pila de llamadas ... hasta alcanzar la función *main*. (en esta función no se pueden seguir lanzando las excepciones, deben capturarse). Veamos un ejemplo:

```
public static void a() throws InterruptedException
{
    throw new InterruptedException ("Exception in a");
}

public static void b() throws InterruptedException
```

```
{
    a();
}

public static void c() throws InterruptedException
{
    b();
}

public static void d() throws InterruptedException
{
    c();
}

public static void main(String[] args)
{
    try
    {
        d();
    } catch (InterruptedException e) {
        System.err.println("Exception: " + e.getMessage());
    }
}
```

Este ejemplo genera una `InterruptedException` en la función `a` (se pueden crear excepciones lanzando nuevos elementos de cualquier tipo de excepción). Se puede observar como la función `b` llama a la función `a`. Esta función es preguntada si va a capturar la excepción o lanzarla. Al añadir la clausula `throws` en la definición de la función, se marca explícitamente esta función para lanzar las excepciones de tipo `InterruptedException`. Esta cadena continúa con las funciones `c` y `d`. Al final, la función *main* llama a la función `d`, y esta función *main* no puede lanzar las excepciones del tipo `InterruptedException`, por lo que es necesario capturar la posible excepción en la función *main*.

Se pueden lanzar (o declarar para ser lanzadas) tantos tipos de excepciones como sea necesario, separadas por comas en la clausula `throws`. Pero antes o después será necesario capturarlas.

```
public static void multipleExceptionsFunction()
throws IOException, InterruptedException
{
    ...
    if (...)
        throw new IOException("IOException produced");
    ...
    if (...)
        throw new InterruptedException("Interrupted!!");
}
...

public static void anotherFunction()
```

```
{
    try
    {
        multipleExceptionsFunction();
    } catch (IOException e1) {
        System.err.println(...);
    } catch (InterruptedException e2) {
        System.err.println(...);
    }
}
```

Toda esta cadena de lanzamiento de excepciones se ha originado en la función `a` , ya que lanza una ***checked exception*** y que necesitamos capturar o lanzar. Si esta función lanzara una ***Runtime exception*** (como `NullPointerException`), entonces ninguno de los `throws` serían necesarios ya que no se trata de una ***checked exception***. El ejemplo sería así:

```
public static void a()
{
    throw new NullPointerException ("Null pointer exception in a");
}

public static void b()
{
    a();
}

public static void c()
{
    b();
}

public static void d()
{
    c();
}

public static void main(String[] args)
{
    d();
}
```

Sin embargo, si intentamos ejecutar este ejemplo, una excepción del tipo `NullPointerException` se producirá en nuestra consola. Como no se trata de una excepción del tipo ***checked***, no es necesario capturarla, pero ya que se produce, deberíamos capturarla para evitar que salga un error en la consola como este:

```

```

```
Exception in thread "main" java.lang.NullPointerException:
Null pointer exception in a
        at Pruebas.a(Pruebas.java:6)
        at Pruebas.b(Pruebas.java:11)
        at Pruebas.c(Pruebas.java:16)
        at Pruebas.d(Pruebas.java:21)
        at Pruebas.main(Pruebas.java:26)
```

Ejercicios propuestos:

2.4.2.1. Crea un programa llamado *CalculateDensity* que pida al usuario un peso (en gramos) y un volumen (en litros) y muestre la densidad. La densidad se calcula dividiendo peso / volumen. El programa debe capturar las excepciones que se puedan producir : `NumberFormatException` y `ArithmeticException` donde se puedan generar. Solo se puede usar el método `Scanner.nextLine` para leer los datos en este ejercicio.

2.4.2.2. Crea un programa llamado *WaitApp* con una función llamada *waitSeconds* que recibirá un número de segundos (entero) como parámetro. Esta función deberá llamar al método `Thread.sleep` para pausar el programa el número de segundos pasados como parámetro (este método funciona con milisegundos, por lo que se deberá convertir los segundos recibidos a milisegundos). Como el método `sleep` puede producir una excepción del tipo `InterruptedException`, se necesitará manejar. En este caso se deberá lanzar la excepción desde la función *waitSeconds* y capturarla en la función *main* que llamará a la función *waitSeconds* con los segundos que habrá recibido como parámetro (en `String[] args`) Después de esperar el número de segundos especificados, el programa mostrará un mensaje de finalización antes de salir.