Rev 1.4 20170209

Programación II

Tema 4. Entrada/Salida

Contenido

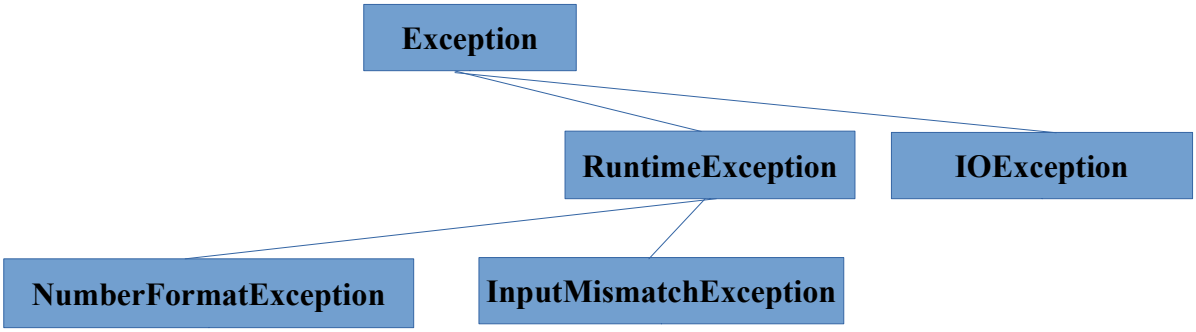
1 Introducción

En este tema se tratan la entrada y salida por consola. Para poder profundizar en los conceptos, se tratan también las clases envolventes de, o asociadas a, los tipos primitivos.

Utilizando excepciones, se podrán crear programas robustos que incluyan entrada y salida por consola (es decir, utilizando el teclado y la pantalla).

2 Excepciones relacionadas

La gestión de excepciones se ha discutido anteriormente. En relación a los temas a tratar, las excepciones que se manejarán son **IOException**, **InputMismatchException** y **NumberFormatException**. Excepto la primera, todas ellas son no controladas.



3 Clases envolventes o asociadas

En Java, es posible utilizar en nuestros programas tipos primitivos como **int**, o **double** o **char**. En realidad, Java es orientada a objetos, por lo que no puede soportar ningún tipo de funcionalidad sobre estos datos tal y como se ofrecen: para poder soportar funciones sobre tipos primitivos, necesita asociarle a cada uno de ellos una clase que se encarga de albergar toda la funcionalidad. En la siguiente tabla se enumeran las clases envolventes para cada tipo.

Tipo primitivo	Clase asociada	Explicación
int	**Integer**	Números enteros.
double	**Double**	Números reales.
char	**Character**	Caracteres Unicode.
boolean	**Boolean**	Booleanos, **true** o **false**.

Así, por ejemplo, la clase asociada a los booleanos es **Boolean**. En la siguiente tabla se listan los métodos principales de esta clase. Nótese que los métodos estáticos pueden ser invocados sin crear, necesariamente, un objeto de la clase.

Miembro	¿Es estático?	Explicación
`TRUE`	Sí	Constante **true**.
`FALSE`	Sí	Constante **false**.
`parseBoolean(s)`	Sí	Transforma una cadena en un booleano.
`toString()`	No	Devuelve el booleano como una cadena.

Igualmente, la clase asociada a los números enteros es **Integer**. En la siguiente tabla se listan los métodos principales de esta clase.

Miembro	¿Es estático?	Explicación
`MAX_VALUE`	Sí	Constante máximo valor entero soportado.
`MIN_VALUE`	Sí	Constante mínimo valor entero soportado.
`parseInt(s)`	Sí	Transforma una cadena en un número entero.
`toString()`	No	Devuelve el número como una cadena.
`doubleValue()`	No	Devuelve el entero como **double**.
`intValue()`	No	Devuelve el valor guardado como **int**.

Como se puede observar en ambos casos, la funcionalidad es muy similar en ambos casos. Concretamente, se ofrece la posibilidad de interpretar una cadena como un valor primitivo, y viceversa, la conversión del valor primitivo a una cadena de texto. Dado que en el primer caso, una cadena puede tener cualquier contenido, los métodos **Integer**.*parseInt(s)*, o **Double**.*parseDouble(s)* pueden lanzar la excepción **NumberFormatException**. A continuación se puede ver un ejemplo[1].

1 El ejemplo se ha obtenido utilizando el *Code Pad* de **BlueJ**.

Ejemplo

```
> int x = Integer.parseInt( “5” );
(int) 5
> Integer intX = new Integer( x );
> intX.toString();
“5”
> double d = Double.parseDouble( “5.6” );
(double) 5.6
> Double doubleD = new Double( d );
> doubleD.toString();
“5.6”
```

Además, es interesante considerar el método estático *format(s, x...)* de la clase **String**. Este método permite crear una cadena a partir de una serie de datos, aportando una plantilla con formato y campos tipo. Los campos tipo más importantes aparecen a continuación.

Campo tipo de formato	Explicación
`%d`	Número entero.
`%f`	Número real.
`%s`	Cadena de texto.

Además, la cadena de formato acepta otros valores que se interpretan de forma diferente a su valor literal.

Constantes de formato	Explicación
`\n`	Cambio de línea.
`\t`	Tabulador (normalmente ocho espacios).
`\\`	Barra invertida.

Por otra parte, los campos tipo numéricos como *%d* o *%f*, aceptan además entre el símbolo '%' y el indicador de formato ('d' o 'f'), un valor numérico que indica el número de caracteres a dedicar al valor a representar. Si el ancho se precede de un 0, se utiliza este dígito para rellenar los espacios sobrantes (por delante), en lugar del espacio. Si se trata de un formato de número real, se puede especificar un segundo valor separado por punto, que indica el número de posiciones decimales.

Sintaxis

```
%{ancho}d
%{ancho}{.posiciones_decimales}f
```

Es necesario tener en cuenta que en el caso de que el valor aportado no se corresponda con el campo tipo, se producirá una excepción **IllegalFormatConversionException** en tiempo de ejecución. Esta excepción es no controlada.

Un ejemplo se muestra a continuación.

```
> String.format( "'%6d'\n", 5 );
'     5'
> String.format( "'%6.2f'\n", 5.6 );
'  5.60'
> String.format( "'%06d'\n", 5 );
'000005'
> String.format( "'%06.2f'\n", 5.6 );
'005.60'
> String asignatura = "PROII";
> String lenguaje = "Java";
> int ano = 2015;
> String.format( "Impartimos %s en %s desde %d.", asignatura, lenguaje, ano );
Impartimos Java en PROII desde 2015.
```

4 Entrada y salida por consola

4.1 Salida por consola

Es muy sencillo mostrar información por la salida estándar (la consola) de Java. Para ello, se utiliza el objeto *out* dentro de la clase **System**. La salida estándar de errores la representa el objeto *err* dentro de **System**. Normalmente, ambas salidas están dirigidas a la pantalla, por defecto. Este objeto tiene varios métodos interesantes, como se ve en la tabla a continuación.

Miembro	Explicación
`print(x)`	Muestra *x* por consola.
`println()`	Cambia de línea.
`println(x)`	Muestra *x* por consola, y cambia de línea.
`format(s, x...)`	Muestra datos por consola, con un formato determinado.

En cuanto al método *format()*, admite como primer argumento una cadena con formato cuyos campos de datos más importantes son exactamente iguales a los del método **String**.*format()*.

Como se ha visto anteriormente, se pueden especificar en los campos tipo para números anchos, que si se preceden con un cero se indica que deben ser rellenados con este dígito. En el caso de los números reales, si se especifica un segundo valor separado por un punto, entonces ese es el máximo número de posiciones decimales. Un ejemplo podría ser el siguiente:

```
> System.out.format( "'%6d'\n", 5 );
'     5'
> System.out.format( "'%6.2f'\n", d );
'  5.60'
> System.out.format( "'%06d'\n", 5 );
'000005'
> System.out.format( "'%06.2f'\n", d );
'005.60'
> String asignatura = "PROII";
> String lenguaje = "Java";
> int ano = 2015;
> System.out.format( "Impartimos %s en %s desde %d.", asignatura, lenguaje, ano );
Impartimos Java en PROII desde 2015.
```

En el siguiente ejemplo, se utiliza el método *main()*, que es la función estática donde siempre arranca la ejecución en un programa en Java. Al utilizar ya *main()*, es posible tanto ejecutarlo seleccionándolo de la lista de métodos en *BlueJ*, como simplemente lanzar la aplicación en *Netbeans, IntelliJ IDEA,* u otro entorno similar.

```
class Persona {
	String nombre;
	int edad;

	public Persona(String n, int e)
	{
		nombre = n;
		edad = e;
	}

	public String getNombre() {
		return nombre;
	}

	public int getEdad() {
		return edad;
	}

	public String toString() {
		return String.format( "%s (%d)", getNombre(), getEdad() );
	}
}
```

```
class Ppal {
      public static void main(String[] args)
      {
            Persona p1 = new Persona( "Baltasar", 21 );
            Persona p2 = new Persona( "Pedro", 18 );
            Persona p3 = new Persona( "Nanny", 19 );

            System.out.println( p1 );
            System.out.println( p2 );
            System.out.println( p3 );
      }
}
```

Salida

```
Baltasar (21)
Pedro (18)
Nanny (19)
```

4.2 Entrada por consola

La entrada por consola estándar se puede realizar fácilmente mediante la clase **Scanner**. Esta clase permite leer fácilmente cadenas desde consola, una vez que se ha creado un **Scanner** aplicado a la entrada estándar. Así, para obtener una nueva cadena de texto leída desde teclado, se utilizará el método *nextLine()*. Tomando la clase **Persona** anterior:

```
import java.util.Scanner;

class Persona { /* ... */ }
```

```
class Ppal {
	public static void main (String[] args)
	{
		String nombre;
		int edad;
		Persona[] v;
		Scanner scan = new Scanner( System.in );
		int maxPersonas = 0;

		// Leer el max. y crear vector
		try {
			maxPersonas = Integer.parseInt( scan.nextLine() );
			v = new Persona[ maxPersonas ];

			// Pedir los datos
			for(int i = 0; i < v.length; ++i) {
				System.out.print( "Dame un nombre: " );
				nombre = scan.nextLine();

				System.out.print( "Dame una edad: " );
				edad = Integer.parseInt( scan.nextLine() );

				v[ i ] = new Persona( nombre, edad );
			}

			// Mostrar los datos
			for(int i = 0; i < v.length; ++i) {
				System.out.println( v[ i ] );
			}
		} catch(NumberFormatException exc)
		{
			System.err.println( "\nERROR: No se ha introducido un número." );
		}
	}
}
```

En realidad, no se ha avanzado demasiado en este ejemplo con respecto a otros, donde se para la ejecución con un mensaje de error, a pesar de realizar un tratamiento de excepciones. Sería interesante tener un método para leer números más robusto, que volviese a pedir el dato en el caso de que no se introdujera un número. Un método como este puede verse a continuación.

```
class Ppal {
      public static int leeNum(Scanner scan, String msg)
      {
            int toret = 0;
            boolean repite;

            do {
                  repite = false;
                  System.out.print( "\n" + msg );

                  try {
                        toret = Integer.parseInt( scan.nextLine() );
                  } catch(NumberFormatException exc)
                  {
                        repite = true;
                  }
            } while( repite );

            return toret;
      }
      //...
}
```

Y por tanto, *main()* puede reescribirse como sigue:

```
class Ppal {
      //...
      public static void main (String[] args)
      {
            String nombre;
            int edad;
            Persona[] v;
            Scanner scan = new Scanner( System.in );
            int maxPersonas = 0;

            // Leer el max. y crear vector
            maxPersonas = leeNum( scan, "Max. personas: " );
            v = new Persona[ maxPersonas ];

            // Pedir los datos
            for(int i = 0; i < v.length; ++i) {
                  System.out.print( "Dame un nombre: " );
                  nombre = scan.nextLine();
                  edad = leeNum( scan, "Dame una edad: " );
                  v[ i ] = new Persona( nombre, edad );
            }

            // Mostrar los datos
            for(int i = 0; i < v.length; ++i) {
                  System.out.println( v[ i ] );
            }
      }
```

En realidad, la clase **Scanner** proporciona varios métodos para lectura de distintos tipos de datos, como por ejemplo el propio **int**, para el que proporciona *nextInt()*. Lógicamente, aún así es posible introducir información que no se corresponda con un entero, por lo que se puede producir la excepción **InputMismatchException**. Así, se puede escribir una variante de *leeNum()* como sigue.

```
public class Ppal {
	//...
	public static int leeNum2(Scanner scan, String msg)
	{
		int toret = 0;
		boolean repite;

		do {
			repite = false;
			System.out.print( "\n" + msg );

			try {
				toret = scan.nextInt();
			} catch(InputMismatchException exc)
			{
				repite = true;
			}
			finally {
				scan.nextLine();
			}
		} while( repite );

		return toret;
	}
}
```

Es necesario llamar a *nextLine()* cada vez que se lea un número, para evitar que el cambio de línea que se queda en el escáner de entrada ('\n') afecte a subsiguientes lecturas. También será necesario para volver a dejar el escáner funcionando llamar a *nextLine()* tras una excepción. Puede probarse a cambiar las llamadas a *leeNum()* por *leeNum2()*.