7. Otras sentencias

Objetivos:

a) Describir el funcionamiento de las otras sentencias de control (break, continue y try-catch)
b) Interpretar el resultado de una secuencia de estas sentencias de control combinadas o no
c) Codificar una tarea sencilla convenientemente especificada, utilizando la secuencia y/o combinación adecuada de estas sentencias de control

7.1. Sentencia break

La sentencia `break` puede encontrarse en sentencias `switch` o en bucles. Al ejecutarse, deja el ámbito de la sentencia en la que se encuentra y pasa a la siguiente sentencia. Puede emplearse con etiquetas, especificando sobre qué sentencia se aplica si hay varias anidadas.

```
etiqueta: sentencia;
break [etiqueta];
```

En la Figura 7.1 se muestra el diagrama de flujo de un bucle `while` que contiene una sentencia `break`:

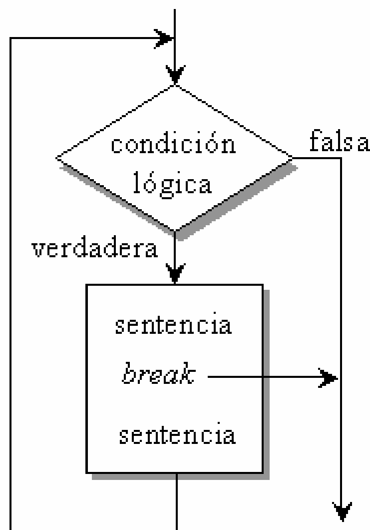

Figura 7.1. Diagrama de flujo de un bucle `while` que incluye una sentencia `break`

El siguiente ejemplo muestra cómo utilizar la sentencia `break` dentro de un bucle `for`:

```
/**
 *  Ejemplo de sentencia break
 *  A. Garcia-Beltran - marzo, 2008
 */
public class TablaProducto2 {
   public static void main (String [] args) {
   int valor;
   valor = Integer.parseInt(args[0]);
```

```
    System.out.println("Tabla de multiplicar del numero " + valor);
    for (int i=1; i<=10; i++) {
       System.out.println(valor + " * " + i + " = " + valor*i );
       if (i==5) break;
       }
    }
}
```

Ejemplo de ejecución y salida correspondiente por pantalla:

```
$>java TablaProducto2 7↵

Tabla de multiplicar del numero 7
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
```

En el resultado de la visualización por pantalla de la ejecución anterior se observa que las líneas correspondientes a los productos de 7 por valores superiores a 5 no se visualizan ya que al ejecutarse la sentencia `break` se abandona el bucle `for`.

7.2. Sentencia continue

La sentencia `continue` se emplea sólo en bucles. Al ejecutarse la iteración en la que se encuentra, el bucle finaliza y se inicia la siguiente. También puede emplearse con etiquetas, especificando sobre que sentencia se aplica si hay varias anidadas.

```
etiqueta: sentencia;
continue [etiqueta];
```

En la Figura 7.2 se muestra el diagrama de flujo de un bucle `while` que contiene una sentencia `continue`:

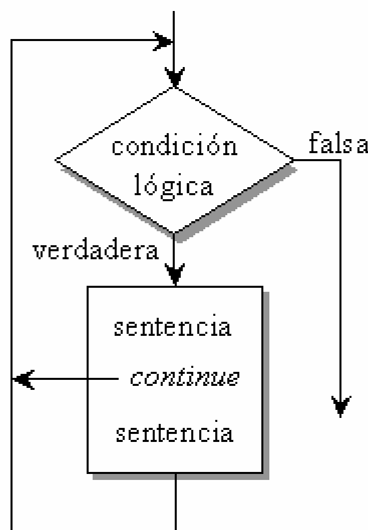

Figura 7.2. Diagrama de flujo de un bucle `while` que incluye una sentencia `continue`

El siguiente ejemplo muestra cómo utilizar la sentencia `continue` dentro de un bucle `for`:

```
/**
 *  Ejemplo de sentencia continue
 *  A. Garcia-Beltran - marzo, 2008
 */
public class TablaProducto3 {
   public static void main (String args [] ) {
   int valor;
   valor = Integer.parseInt(args[0]);
   System.out.println("Tabla de multiplicar del numero " + valor);
   for (int i=1; i<=10; i++) {
      if (i==5) continue;
      System.out.println(valor + " * " + i + " = " + valor*i );
      }
   }
}
```

Ejemplo de ejecución y salida correspondiente por pantalla:

```
$>java TablaProducto3 7↵

Tabla de multiplicar del numero 7
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
```

En el resultado de la visualización por pantalla de la ejecución anterior se observa que la línea correspondiente al producto de 7 por 5 es la única que no se visualiza.

Ejemplo de utilización combinada de `break` y `continue` con y sin etiquetas:

```
uno: for( ... ; ... ; ... )
  {
  dos: for( ... ; ... ; ... )
    {
    // Grupo de sentencias A
    sentenciasA;
    continue;     // sigue en el bucle interno
    // Grupo de sentencias B
    sentenciasB;
    continue uno; // sigue en el principal
    // Grupo de sentencias C
    sentenciasC;
    break uno;    // sale del principal
    // Grupo de sentencias D
    sentenciasD;
    }
  }
```

7.3. Tratamiento de excepciones

La ejecución de determinadas sentencias puede dar lugar a la generación de problemas o excepciones. Para gestionar una excepción debe emplearse una sentencia `try`. La sintaxis de la sentencia se muestra a continuación:

```
try {
     sentencia_1;
     sentencia_2;
    ...
}
catch (claseExcepcion objetoExcepcion) {
     sentencia_a;
     sentencia_b;
     ...
}
```

La sentencia comienza por la palabra reservada `try` seguida de una o más sentencias agrupadas entre paréntesis. Dichas sentencias son las que, en principio, pueden dar lugar a una excepción durante su ejecución. La clase `Exception`, que incluye todos los tipos de excepciones de interés, recoge cualquier excepción generada por el bloque `try`. Por ejemplo, determinados métodos como `readLine` generan una excepción si se produce un problema con la entrada o salida de datos. En este caso, se necesita recoger las excepciones de tipo `IOException`. La ejecución de otros metódos, como `parseInt`, genera un error si la cadena dada como parámetro no puede convertirse en un entero. En este caso, la excepción es de tipo `NumberFormatException`.

A continuación aparece una o más clausulas `catch` que son las *manejadoras* de las excepciones. En cada una de ellas, se necesita especificar lo que hacer cuando ocurre una excepción en particular. Esta parte del código sólo se ejecuta si se ha producido una excepción.

Programa sencillo:

```
/**
 *  Demostracion de excepcion
 *  A. Garcia-Beltran - marzo, 2008
 */
import java.io.*;             // Importa todas las clases del paquete java.io
public class RaizCuadradaEntero {
   public static void main (String [] args) {
   // Permite la entrada de datos por la entrada estandar
   BufferedReader in = new BufferedReader ( new
                            InputStreamReader( System.in ) );
   int valor;
   String cadena;
   System.out.print("Por favor, introduce un entero positivo: ");
   try {
       cadena = in.readLine();
       valor = Integer.parseInt(cadena);
       System.out.println("Raiz cuadrada = " + Math.pow(valor,0.5));
       }
   catch (Exception exc )
       { System.out.println( exc ); }
   System.out.println("Esta es la sentencia final del programa. ");
   }
}
```

En el código anterior, se imprime un mensaje significativo para el objeto `exc` de tipo `Exception`. Como alternativa, podría realizarse un procesamiento adicional o bien darse mensajes de error más detallados. Ejemplo de ejecución y salida por pantalla:

```
$>java RaizCuadradaEntero↵
Por favor, introduce un entero positivo: 4↵
La raiz cuadrada es 2.0
Esta es la sentencia final del programa.
```

Otros ejemplos de ejecución y salidas correspondientes por pantalla:

```
$>java RaizCuadradaEntero↵
Por favor, introduce un entero positivo: 4.0↵
java.lang.NumberFormatException: For input string: "4.0"
Esta es la sentencia final del programa.

$>java RaizCuadradaEntero↵
Por favor, introduce un entero positivo: 10000000000↵
java.lang.NumberFormatException: For input string: "10000000000"
Esta es la sentencia final del programa.

$> java RaizCuadradaEntero↵
Por favor, introduce un entero positivo: cuatro
java.lang.NumberFormatException: For input string: "cuatro"
Esta es la sentencia final del programa.

$> java RaizCuadradaEntero↵
Por favor, introduce un entero positivo: -4
Raiz cuadrada = NaN
Esta es la sentencia final del programa.
```

En Java existen muchos tipos de excepciones estándar. Algunas de las excepciones más comunes se muestran en la Tabla 7.1.

Tabla 7.1. Excepciones y significados

Excepción	Significado
`IOException`	Problema de entrada o salida de datos
`ArithmeticException`	Desbordamiento o división entera por cero
`NumberFormatException`	Conversión ilegal de un string a un tipo numérico
`IndexOutOfBoundsException`	Acceso a un elemento inexistente de un vector o de un `String`
`NegativeArraySizeException`	Intento de creación de un vector de longitud negativa
`NullPointerException`	Intento de uso de una referencia nula
`SecurityException`	Violación de la seguridad en tiempo de ejecución

7.4. Operaciones de entrada y salida de datos

Como se ha mostrado en el ejemplo anterior las operaciones de entrada y salida de datos en un programa se llevan a cabo utilizando el paquete `java.io`. La sentencia `import java.io.*;` da acceso a la librería de Java necesaria para cualquier operación de este tipo. Los canales o dispositivos predefinidos para realizar entradas o salidas de datos son los siguientes:

- `System.in`: entrada estándar
- `System.out`: salida estándar

- `System.err`: salida de errores

Los métodos `print` y `println` se emplean para la salida de datos en formato de concatenación de `Strings`. El método `readLine` facilita una forma sencilla para realizar la entrada de datos mediante un objeto `String`. Este objeto toma el valor de la cadena de caracteres que acaben en un final de línea o en un final de archivo. Para poder emplear el método `readLine` es necesario construir un objeto `BufferedReader` sobre un objeto `InputStreamReader`, que a su vez se crea a partir de `System.in`.