5

Estructuras de control

Objetivos

- ✓ Conocer los mecanismos básicos de entrada y salida de datos desde consola en un programa.
- ✓ Comprender el manejo de las diferentes estructuras de selección, repetición y salto.
- ✓ Desarrollar un programa que evalúe datos de entrada y devuelva el resultado correspondiente.

Mapa conceptual

Glosario

Argumento. Valor variable pasado a una función o método (sirva de ejemplo el método main) para que realice una tarea en función de este.

Array. Almacenamiento en memoria de un conjunto de datos ordenado del mismo tipo. También se conoce como vector o matriz de una dimensión.

Bug. Término genérico que define aquel comportamiento que tiene un programa y para el que no fue diseñado.

Iteración. En programación se denomina *iteración* a cada repetición de una sentencia o conjunto de ellas.

Parámetro. Variable declarada en la misma declaración de una función o método (sirva de ejemplo el método *main*), que recoge el valor de los argumentos con los que se realizó la invocación.

Password. Contraseña, código de acceso o clave de paso que es necesario conocer para acceder a una determinada información o sistema.

Sentencia condicional. Sentencia que transfiere el control del programa a otra línea de ejecución en función de una o varias condiciones.

Sentencia incondicional. Sentencia que transfiere el control del programa a otra línea de ejecución sin tener en cuenta ninguna condición.

Sentencia. Instrucción perteneciente a un lenguaje de programación que es ejecutada en un programa informático.

Stream. Flujo de datos empleado para la salida y entrada de información.

String. Secuencia de caracteres Unicode que, a diferencia de otros lenguajes, en Java es representada mediante una clase de mismo nombre que facilita su manejo.

Token. Cadena de caracteres que tiene un significado especial y que viene contenida en una cadena de mayor tamaño.

5.1. Introducción

Los programas son de muy diverso tipo y, debido a ello, pueden tomar los datos con los que trabajan desde varios orígenes; además, también son diseñados para mostrar los resultados de distintas maneras. Pero independientemente de cómo obtengan los datos con los que trabajan, la ejecución de las instrucciones contenidas en ellos es secuencial y sigue un orden que va desde las primeras líneas al final del programa. Gracias a las sentencias de control será posible alterar el orden de ejecución según la lógica que sea necesario aplicar.

En este capítulo se explicará la forma básica de comunicación con un programa Java, así como las sentencias de control que permiten trabajar con la información recibida.

5.2. Entrada y salida de información

Existen varias alternativas para comunicarse con los programas Java. En los apartados siguientes se explicarán la entrada de información por argumentos (en la llamada) y la entrada-salida de información por el terminal.

5.2.1. Entrada por argumentos

Todo programa Java tiene como punto de partida el método *main* declarado en la clase principal. A partir de ese método el programa realizará las tareas correspondientes hasta que finalice el trabajo o se detenga, bien por causas controladas o ajenas a la normal ejecución de este.

El programa *main* se declara como un método estático (no necesita de una instancia de la clase que lo contiene para poder ser invocado), público (es accesible desde cualquier lugar) y con el parámetro *args* que contendrá los argumentos pasados cuando el programa se invoque con argumentos. Dicho parámetro es del tipo *array*, un tipo complejo no visto hasta ahora que se corresponde con una colección de elementos y que será explicado en capítulos posteriores.

Hasta ahora únicamente se han ejecutado programas que establecían los datos con los que trabajan desde el mismo programa y sin ningún paso de argumentos o interacción con el usuario.

PARA SABER MÁS

Los términos *parámetro* y *argumento* a menudo se utilizan indistintamente, pero en realidad no son sinónimos. Un argumento es el dato que se pasa cuando se hace la llamada a un método, mientras que un parámetro es la variable en la que se almacena el dato recibido desde la invocación.

Para pasar argumentos a un programa es necesario añadirlos a la llamada (tanto si se ejecutó de forma simple como si venía contenido en un *jar* ejecutable). Podrán enviarse tantos argumentos como sea necesario, cada uno de ellos separado por un espacio. Dentro del método *main* podrán recuperarse como *Strings* con *args[0]* para el primer argumento, *args[1]* para el segundo… y así sucesivamente.

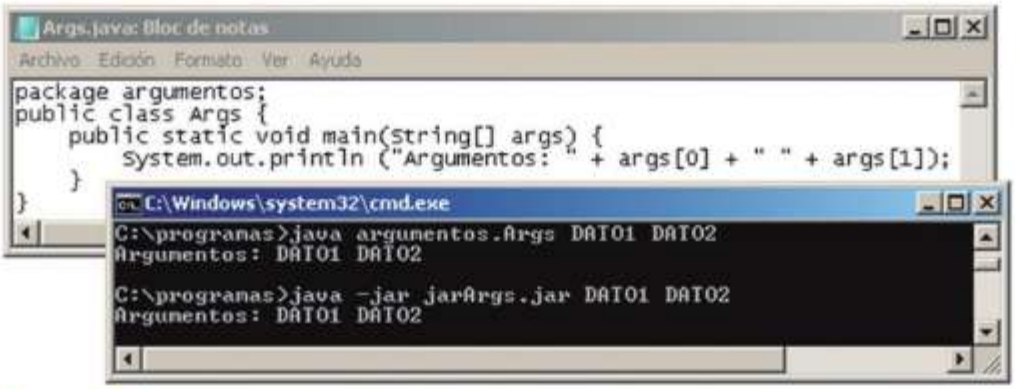

Figura 5.1
Paso de información por argumentos desde línea de comandos.

Si se desea trabajar desde *NetBeans* con argumentos, hay que acceder a las *propiedades del proyecto*, categoría *Run* y en la sección *Arguments* especificar los argumentos de la misma forma que se haría por el terminal.

Actividad propuesta 5.1

¿Es posible enviar la frase "Hola mundo" con un único argumento a un programa Java desde la terminal? ¿Y desde *NetBeans*? Justifica tu respuesta.

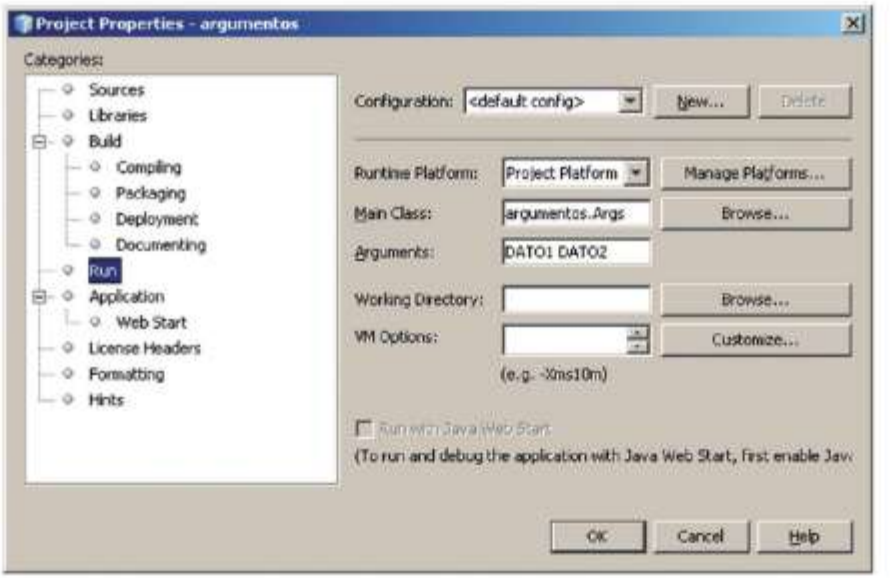

Figura 5.2
Paso de información por argumentos desde *NetBeans*.

5.2.2. Entrada y salida por consola

Mediante la invocación con argumentos se puede pasar información al programa que podrá recoger para comenzar la ejecución. Sin embargo, solo con esto no será posible implementar programas interactivos. En Java la entrada y salida de datos se realiza mediante flujos de datos (*Streams*). Este mecanismo, que ha estado presente desde la primera versión de Java, proporciona una serie de flujos de datos estándar que estarán disponibles mientras el programa esté en ejecución y listos para proporcionar datos. Son los siguientes:

1. *Flujo estándar de salida System.out*: se trata de un objeto de la clase *PrintStream* que por defecto es la pantalla. Su manejo no presenta ninguna complicación, dado que ya ha sido utilizada en ejemplos anteriores. Para mostrar datos por la consola se puede hacer uso de los métodos:

   a) *System.out.write (datos)* → Imprime los *bytes* enviados como argumento por el flujo estándar.
   b) *System.out.print (obj)* → La información que se le envía como argumento se transforma a un conjunto de *bytes* y posteriormente se imprime haciendo uso del método *System.out.write*.
   c) *System.out.println (obj)* → Igual que el anterior añadiendo un salto de línea al final.
   d) *System.out.printf (String cadena, [tipo argumento1] … [,tipo argumentoN])* → Imprime *cadena* incluyendo los *argumentos* en las especificaciones de formato. En el caso de que el número de argumentos sea superior al de especificaciones de formato en cadena, los argumentos sobrantes se ignoran.

2. *Flujo estándar de salida de error System.err*: similar a *System.out* pero para salida de error.

3. *Flujo estándar de entrada System.in*: se trata de un objeto de la clase *InputStream* que por defecto es el teclado. Para leer datos desde el teclado se puede hacer uso de los métodos:

   a) *System.in.read (datos)* → Almacena en *datos* los caracteres leídos desde el teclado. Datos debe ser declarado previamente como un *array* de *bytes*.
   b) *System.in.read ()* → Retorna el *byte* siguiente de datos desde el flujo de entrada como un entero comprendido entre 0 a 255 (se debe convertir a *char* mediante un *casting*).

A continuación, puede verse un ejemplo de Entrada/Salida mediante flujos estándar:

```
int c1, c2, c3;
System.out.println("Introduzca 3 caracteres:");
c1 = System.in.read();
c2 = System.in.read();
c3 = System.in.read();
System.out.write(c1);
System.out.print((char)c2);
System.out.println((char)c3);
```

Otro mecanismo mucho más cómodo para realizar la lectura de información desde un *Stream* es la clase *Scanner*. Esta clase, que fue incluida a partir de *Java 1.5*, es capaz de analizar tipos primitivos y dividir la entrada en *tokens* que, por defecto, vienen separados por espacios. Algunos de los métodos más conocidos son:

- ✓ *nextLine ()* → Retorna la línea completa como un *String*.
- ✓ *next ()* → Retorna el siguiente *token* como un *String*.
- ✓ *nextXXXX ()* → Retorna el siguiente *token* como si fuese de tipo (siendo *XXXX Byte, Int, Double…*).
- ✓ *hasNext ()* → Devuelve *true* si el escáner puede devolver otro *token*.
- ✓ *hasNextInt ()* → Devuelve *true* si el próximo *token* puede interpretarse como un valor de tipo *XXXX* utilizando el método *nextXXXX()*.

A continuación, puede verse un ejemplo de entrada/salida utilizando la clase *Scanner*:

```
package entradasalida ;
import java.util.Scanner;
public class EntradaSalida {
    public static void main(String[] args) {
         System.out.println("Introduzca el texto a reproducir:");
         Scanner entradaEscaner = new Scanner (System.in);
         String entradaTeclado = entradaEscaner.nextLine ();
         System.out.println (entradaTeclado);
    }
}
```

También puede ser de gran utilidad la clase *Console* que incluye funcionalidad para lectura de *passwords* ocultando los caracteres tecleados. Está disponible desde *Java 1.6* y sus métodos principales son:

- *readLine ()* → Retorna la línea completa como un *String*.
- *readPassword ()* → Devuelve un *array* de caracteres pero ocultando los caracteres que son introducidos por teclado.
- *printf (String cadena, [tipo argumento1] … [,tipo argumentoN])* → Imprime texto por consola de forma similar a *System.out.printf*.

En la figura 5.3 puede verse el resultado de ejecutar el código del siguiente ejemplo en el que se utiliza la clase *Console* para la entrada y salida de datos.

```
package entradasalida;
import java.io.Console;
public class EntradaSalida {
    public static void main(String[] args) {
        Console c = System.console();
        if (c == null)
            System.err.println("No hay consola.");
        else {
            System.out.println("Introduzca su nombre: ");
            String nombre = c.readLine();
            System.out.println("Hola "+nombre);
            System.out.println("Introduzca su password: ");
            String password = String.valueOf(c.readPassword());
            System.out.println("Su Password es: "+password);
        }
    }
}
```

```
C:\Windows\system32\cmd.exe
C:\Users\[redacted]\Documents\NetBeansProjects\entradasalida\build\classes>java entr
adasalida.EntradaSalida
Introduzca su nombre:
Pedro
Hola Pedro
Introduzca su password:

Su Password es: Síntesis1

C:\Users\genete\Documents\NetBeansProjects\entradasalida\build\classes>_
```

Figura 5.3
Ejemplo de Entrada/Salida utilizando la clase *Console*.

Recurso web

En la URL https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax puedes encontrar más información sobre los especificadores de formato para cadenas Java.

Actividades propuestas

5.2. Busca información sobre el método *System.out.flush* e indica un caso en el que pueda ser de utilidad.

5.3. Al método *next* de la clase *Scanner* se le puede enviar como argumento una expresión regular que sea la que utilice para separar los *tokens*. Investiga sobre cómo se forman las expresiones regulares y construye algunas para reconocer los patrones de un DNI, número de teléfono y dirección de correo electrónico.

5.3. Estructuras de selección (alternativa)

Todo lenguaje de programación ofrece estructuras de control y selección con las que escribir el código. A continuación, se van a exponer las sentencias de control que son la materialización en el lenguaje Java de las estructuras de control expuestas en el capítulo 1 (Introducción a la programación).

5.3.1. *if, else, else if*

La sentencia condicional *if* es la sentencia de control más básica de cualquier lenguaje de programación. Permite evaluar una condición y actuar en consecuencia en función de su resultado. Existen algunas variantes de esta sentencia condicional:

1. *if* → Se evalúa la condición encerrada entre paréntesis y si el valor de retorno es *true*, se ejecutará el bloque de instrucciones encerrado entre llaves. En caso contrario se saltará a la siguiente instrucción.
2. *if-else* → Similar a la sentencia *if* anterior, pero en este caso también es capaz de ejecutar un bloque de instrucciones si la condición evaluada retorna *false*.
   Además de las sentencias anteriores hay que destacar la existencia del operador ternario que permite realizar un *if-else* simple en una única línea de código.
3. *if-else if* → La sentencia *if-else* ejecuta un segundo bloque de instrucciones en el caso de que la condición inicial devolviera *false* y esto lo hacía sin evaluar nuevas condiciones. Si se necesita seguir evaluando condiciones una vez que la primera no se ha cumplido, es necesario hacer el *else* condicional. Para ello se irán encadenando diferentes bloques *if-else*.

Cada bloque de instrucciones puede contener a su vez nuevas sentencias condicionales. De esta manera se pueden ir anidando unos a otros. A continuación pueden verse varios ejemplos de sentencias condicionales *if*:

	EQUIVALENTES ↓	

`if (a>b) {`
`  mayor = a;`
`  menor = b;`
`}`
	`if (x>y)`
	`  mayor = x;`
	`else`
	`  mayor = y;`
		`if (dia=="lunes") {`
		`  …`
		`} else if (dia=="martes") {`
		`  …`
		`} else if (dia=="domingo") {`
		`  …`
		`} else {`
		`  …`
		`}`

	`mayor = (x>y)?x:y;`	

5.3.2. *Switch*

Muchas de las soluciones que se implementan con la sentencia *if-else if* pueden realizarse de forma más clara con *switch*. La sentencia condicional *switch* permite evaluar una variable y ejecutar un bloque de sentencias en función del valor que tome. Puede incluirse, es opcional, el valor *default* que será el que determinará las instrucciones a ejecutar si no hubo coincidencia con ninguno de los casos anteriores. Esta sentencia de control es capaz de trabajar con:

- ✓ Tipos primitivos *byte*, *short*, *char*, *int*.
- ✓ Tipos *enumerados*.
- ✓ *Strings*.
- ✓ Clases envoltorio *Character*, *Byte*, *Short* e *Integer*.

A continuación, puede verse un ejemplo de uso de sentencias condicionales *switch* en el que se muestran los días que faltan hasta el fin de semana:

```
int dia = 1; // 1-L, 2-M, 3-X, 4-J, 5-V
switch (dia) {
   case 1:  System.out.println("Lunes");
   case 2:  System.out.println("Martes");
   case 3:  System.out.println("Miércoles");
   case 4:  System.out.println("Jueves");
   case 5:  System.out.println("Viernes");
            break;
   default: System.out.println("Fin de semana");
            break;
}
```

Como se habrá podido comprobar, en los bloques de instrucciones de cada opción es posible encontrar un *break*. Esta instrucción provoca una interrupción en el flujo de ejecución del bloque en el que se encuentre y fuerza a que se ejecute la instrucción siguiente al bloque interrumpido. Si no se introduce esta instrucción, se seguirán ejecutando las instrucciones que encuentre a continuación. La instrucción *break* se verá en detalle en apartados posteriores.

5.4. Estructuras de repetición (iterativa)

Las estructuras de repetición, o bucles como suelen denominarse en jerga informática, permiten ejecutar bloques de instrucciones durante un número determinado o indeterminado de iteraciones.

5.4.1. *for*

La sentencia de control *for* proporciona una forma de iterar sobre un determinado rango de valores. Está formada por una *expresión de inicialización* (que únicamente se ejecutará al comienzo), una *expresión de finalización* que causará la finalización del bucle cuando retorne *false* y una *expresión de incremento/decremento* que es invocada cada *iteración* del bucle. Si se omiten las 3 expresiones anteriores, será un bucle infinito. A continuación, se muestran varios ejemplos de sentencias *for*:

```
for (int i=1; i<4; i++) {
    System.out.println("Incrementando: " + i);
}
for (int j=3; j>0; j--) {
    System.out.println("Decrementando: " + j);
}
for ( ; ; ) {
    System.out.println("Bucle infinito");
}
```

```
Incrementando 1
Incrementando 2
Incrementando 3
Decrementando 3
Decrementando 2
Decrementando 1
Bucle infinito
Bucle infinito
...
```

La declaración de la variable empleada en la *expresión de incremento/decremento* podrá realizarse fuera de la sentencia iterativa o bien estar contenida de forma implícita en la *expresión de inicialización* (tal como se ha hecho en los ejemplos). La diferencia está en que, si se declara fuera la variable, esta seguirá existiendo hasta que finalice el bloque en el cual ha sido declarado el *for*, mientras que, si se hace en el propio *for*, se destruirá una vez finalice el bucle:

```
public static void main(String[] args) {
    int i=0;
    for (i=0; i<4; i++) {}
    System.out.println("Valor de i: " + i);
}
```

```
run:
Valor de i: 4
```

5.4.2. *while*

La sentencia *while* evalúa la condición y si retorna *true* ejecuta el bloque de instrucciones. Las sentencias contenidas puede que no se ejecuten nunca si la condición inicialmente retorna un

valor *false*. Salvo que se quiera implementar un bucle infinito, las instrucciones contenidas en el bloque deberán alterar el resultado de la condición en algún momento. En el ejemplo siguiente puede verse cómo utilizar una sentencia *while* y el resultado producido:

```
int i = 1;
while (i < 4) {
    System.out.println("Iteración: " + i);
    i++;
}
```

```
Iteración: 1
Iteración: 2
Iteración: 3
```

5.4.3. *do while*

Esta sentencia es parecida a *while* pero con la diferencia de que la sentencia *do while* primero ejecuta el bloque de instrucciones y posteriormente evalúa la condición. Por tanto siempre ejecutará el bloque de instrucciones como mínimo una vez. A continuación se muestra un ejemplo de uso de la sentencia *do while* y del resultado que produciría:

```
int i = 1;
do {
    System.out.println("Iteración: " + i);
    i++;
} while (i < 1);
```

```
Iteración: 1
```

5.5. Estructuras de salto incondicional

Este tipo de estructuras de control permiten modificar el flujo normal de ejecución el programa para realizar saltos a diferentes partes del código fuente. Aunque son expuestas en este apartado, su uso está totalmente desaconsejado desde el punto de vista de la programación estructurada, puesto que complica el seguimiento y trazabilidad de los programas. Este tipo de estructuras son perfectamente reemplazables por las expuestas anteriormente (*secuencial*, *alternativa* e *iterativa*).

5.5.1. *break continue*

La sentencia *break* permitirá detener la ejecución de un bucle y saltar a la siguiente instrucción del programa tras el bucle. En los ejemplos siguientes puede verse cómo utilizar la sentencia de salto *break* dentro de una sentencia *while*:

```
while (true) {
    System.out.println("Dentro bucle 1");
    while (true) {
        System.out.println("Dentro bucle 2");
        break;
    }
}
```

```
Dentro bucle 1
Dentro bucle 2
Dentro bucle 1
Dentro bucle 2
.
.
.
```

```
int j=1;
while (true){
    System.out.println("Iteración " + j);
    if (j==3) break;
    j++;
}
System.out.println("Tras 3 iteraciones salgo");
```

```
Iteración 1
Iteración 2
Iteración 3
Tras 3 iteraciones salgo
```

La sentencia *continue* permite detener únicamente la *iteración* actual y pasar a la siguiente *iteración* del bucle sin salir de él (salvo que el propio bucle haya llegado a su fin). En los ejemplos mostrados a continuación podrá verse cómo utilizar la sentencia de salto *continue*:

```
while (true) {
    System.out.println("Dentro bucle 1");
    while (true) {
       System.out.println("Dentro bucle 2");
       continue;
    }
}
```

```
Dentro bucle 1
Dentro bucle 2
Dentro bucle 2
Dentro bucle 2
.
.
.
```

```
for(int j=1; j<=3; j++){
    System.out.println("Iteración " + j);
    if (j<3) continue;
    System.out.println("Saliendo del bucle");
 }
System.out.println("Fuera del bucle");
```

```
Iteración 1
Iteración 2
Iteración 3
Saliendo del bucle
Fuera del bucle
```

Normalmente, las instrucciones *break* y *continue* van dentro de una sentencia *if* que indicará cuándo detener el bucle en función de una condición. La sentencia *break* puede encontrarse dentro de un bucle o en una sentencia *switch*, mientras que la sentencia *continue* debe ir siempre dentro de un bucle.

5.5.2. *break continue* con etiquetas

Tanto *break* como *continue* pueden usar etiquetas para que, en lugar de salir del bucle o saltar a la siguiente *iteración* como ocurre cuando se usan sin ninguna etiqueta, continúen por la sección

del código identificada por la etiqueta. Las etiquetas no pueden colocarse en cualquier parte del código, por lo que únicamente se podrán hacer saltos a bucles dentro de los que nos encontremos. Las etiquetas se declaran como *nombreEtiqueta:*. A continuación, se muestra un ejemplo de uso de estas junto con sentencias de salto *break-continue*:

```
A: while (true) {
    System.out.println("Dentro de A");
    B: while (true) {
        System.out.println("Dentro de B");
        break A;
    }
}
C: while (true) {
    System.out.println("Dentro de C");
    D: while (true) {
        System.out.println("Dentro de D");
        continue C;
    }
}
```

```
Dentro de A
Dentro de B
Dentro de C
Dentro de D
Dentro de C
Dentro de D
Dentro de C
Dentro de D
Dentro de C
Dentro de D
Dentro de C
.
.
.
```

Actividad propuesta 5.4

Entre las palabras reservadas del lenguaje Java que fueron expuestas en el apartado 4.3 se encuentra *goto*. Busca información sobre ella y propón, si es viable, un ejemplo de uso.

5.6. Prueba y depuración de programas

Todo programa deberá ser testado mediante un conjunto de pruebas que garanticen que este sea operativo, cumpla con los requisitos para los que fue desarrollado e incluya comentarios lo suficientemente aclaratorios como para facilitar su comprensión a otros desarrolladores. Ligado a este proceso de pruebas está el término *bug*, el cual define al comportamiento anómalo de un programa (para el que este no fue diseñado).

Al proceso mediante el cual se corrigen estos errores del programa se le conoce como depuración e implica una serie de tareas:

- La inspección del código del programa o algoritmo mediante el seguimiento del flujo del mismo, intentando predecir los resultados cuando esto sea posible.
- La utilización de baterías de prueba que sometan al programa a un trabajo exigente, de manera que ante una ejecución exitosa se pueda pronosticar un correcto funcionamiento en un escenario real.
- La prueba de todos y cada uno de los módulos de los que pueda constar el programa (pruebas unitarias).
- La corrección de todos y cada uno de los errores detectados, teniendo en cuenta que ante cada modificación del código pueden surgir nuevos errores.

INVESTIGA

JUnit y *TestNG* son dos herramientas que se pueden emplear para llevar a cabo pruebas unitarias de los diferentes módulos de un programa. Investiga cómo hacer uso de ellas sobre el IDE NetBeans.

Resumen

- Un programa puede recibir datos de entrada a través de sus parámetros o bien a través de flujos de entrada.
- Por defecto, todo programa dispone durante su ejecución de 2 flujos de salida (normal y errores) y uno de entrada.
- Existen diferentes sentencias de control (*if* y *switch*), de repetición (*for*, *while* y *do while*) y de salto incondicional (*break* y *continue*), estas últimas desaconsejadas.

Ejercicios propuestos

1. Describe los métodos empleados para imprimir y leer información desde consola.
2. ¿Es posible separar la salida estándar de la salida de error? Razona tu respuesta.
3. ¿Por qué motivo no se recomienda el uso de estructuras de salto incondicional?
4. Compara las instrucciones *break* y *continue*. Propón un ejemplo.
5. Realiza un estudio sobre los diferentes métodos *next* de la clase *Scanner* y propón ejemplos de uso.

Supuestos prácticos

1. Implementa un programa que pida al usuario por teclado sus datos personales en orden *apellido1-apellido2-nombre* y los muestre por pantalla en orden *nombre-apellido1-apellido2*.
2. Desarrolla un programa que, una vez ejecutado, lea información desde teclado hasta que reciba la cadena de entrada "FIN".
3. Implementa un programa que reciba como parámetro un dígito correspondiente a un año y calcula si es o no bisiesto.
4. Escribe un programa calculadora que reciba como parámetros dos enteros (*num1* y *num2*) desde su llamada y que los muestre por pantalla. A continuación, se deberá solicitar al usuario

que seleccione la operación matemática a realizar entre los dos números. Las operaciones que implementará serán suma, resta, multiplicación, división y potencia.

5. Desarrolla un programa que pregunte por pantalla un número N y que, una vez leído, imprima una pirámide N filas.

6. Utilizando la sentencia de control *switch*, implementa un programa que pida un número por teclado y que indique por pantalla a qué mes se corresponde.

7. Analiza el siguiente fragmento de código y describe su comportamiento:

```
System.out.println("Introduzca un texto: ");
while (true){
     int letra = System.in.read();
     System.out.print((char)letra);
}
```

ACTIVIDADES DE AUTOEVALUACIÓN

1. ¿Cuál de las siguientes afirmaciones es falsa?
   - ☐ *a)* A un *jar* ejecutable se le envían los argumentos separados por espacios.
   - ☐ *b)* A un programa Java se le envían los argumentos separados por espacios.
   - ☐ *c)* Los parámetros que recibe un programa pueden ser de cualquier tipo primitivo.
   - ☐ *d)* El primer parámetro recibido se corresponde con *args[0]*.

2. ¿Qué instrucción imprime un salto de línea retorno de carro al final del texto que recibe?
   - ☐ *a)* *System.out.write*.
   - ☐ *b)* *System.out.print*.
   - ☐ *c)* *System.out.println*.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

3. ¿Cuál de los siguientes métodos admite especificaciones de formato?
   - ☐ *a)* *System.out.write*.
   - ☐ *b)* *System.out.printf*.
   - ☐ *c)* *System.out.print*.
   - ☐ *d)* *System.out.println*.

4. ¿Cuál de las siguientes no representa un objeto de la clase *PrintStream*?
   - ☐ *a)* *System.in*.
   - ☐ *b)* *System.out*.
   - ☐ *c)* *System.err*.
   - ☐ *d)* Niguna de las anteriores opciones representa objetos de la clase *PrintStream*.

5. ¿Qué clase de las siguientes facilita la lectura de contraseñas ocultando los caracteres introducidos?
   - ☐ *a)* *java.util.Scanner.*
   - ☐ *b)* *java.io.Console.*
   - ☐ *c)* *java.io.Scanner.*
   - ☐ *d)* *java.util.ConsoleSystem.out.*

6. ¿Con cuál de las siguientes sentencias se puede realizar un bucle infinito?
   - ☐ *a)* *while.*
   - ☐ *b)* *for.*
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Las opciones *a)* y *b)* son incorrectas.

7. ¿Es posible hacer un bucle infinito con una sentencia do *while*?
   - ☐ *a)* Sí, haciendo que la condición evaluada de *while* siempre sea *true*.
   - ☐ *b)* Sí, haciendo que el bloque *do* que se ejecuta como mínimo una vez sea en sí mismo un bucle infinito.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Las opciones *a)* y *b)* son incorrectas.

8. Si se desea ejecutar un bloque de instrucciones hasta que se cumple una condición, pero que se ejecute como mínimo una vez, se deberá emplear:
   - ☐ *a)* *while.*
   - ☐ *b)* *do while.*
   - ☐ *c)* *for.*
   - ☐ *d)* Todas las opciones anteriores son correctas.

9. ¿Cuál de las siguientes es una afirmación falsa?
   - ☐ *a)* La sentencia *switch* puede llevar sección *default*.
   - ☐ *b)* La sentencia *switch* puede llevar *break* en sus secciones.
   - ☐ *c)* La sentencia *switch* siempre lleva sección *default*.
   - ☐ *d)* En ocasiones una sentencia *switch* puede reemplazar una sentencia *if-else if*.

10. ¿Cuáles son los tipos de datos capaces de manejar una sentencia *switch*?
    - ☐ *a)* Tipos *enumerados*, *Strings* y clases envoltorio *Character*, *Byte*, *Short* e *Integer*.
    - ☐ *b)* Tipos primitivos *byte*, *short*, *char*, *int*, *Strings* y clases envoltorio *Character*, *Byte*, *Short* e *Integer*.
    - ☐ *c)* Tipos primitivos *byte*, *short*, *char*, *int*, tipos enumerados, *Strings*.
    - ☐ *d)* Tipos primitivos *byte*, *short*, *char*, *int*, tipos enumerados, *Strings* y clases envoltorio *Character*, *Byte*, *Short* e *Integer*.

SOLUCIONES:

1.	a	b	**c**	d	**5.**	a	**b**	c	d
2.	a	b	**c**	d	**6.**	a	b	**c**	d
3.	a	**b**	c	d	**7.**	a	b	**c**	d
4.	**a**	b	c	d	**8.**	a	**b**	c	d
					9.	a	b	**c**	d
					10.	a	b	c	**d**