UD03 – ESTRUCTURAS DE CONTROL

Contenido

1 Estructuras de selección

Las estructuras de selección permiten tomar decisiones sobre qué conjunto de instrucciones a ejecutar en un punto del programa. O sea, seleccionar qué código se ejecuta en un momento determinado entre caminos alternativos.

Toda estructura de selección se basa en la evaluación de una expresión que debe dar un resultado booleano: true(cierto) o false(falso). Esta expresión se denomina la condición lógica de la estructura.

El conjunto de instrucciones que ejecutará dependerá del resultado de la condición lógica, actuando como una especie de interruptor que marca el flujo a seguir dentro del programa. Esta condición lógica se basa en parte, o en su totalidad, en valores almacenados en variables con un valor que puede ser diferente para diferentes ejecuciones del programa.

1.1 Selección simple (if)

El caso más simple dentro de las estructuras de selección es aquel en que hay un conjunto o bloque de instrucciones que sólo desea que se ejecuten bajo unas circunstancias concretas. De lo contrario, este bloque es ignorado y, desde el punto de vista de la ejecución del programa, es como si no existiera. Un ejemplo sería el programa de una tienda virtual que aplica un descuento al precio final de acuerdo con un cierto criterio (por ejemplo, si la compra total es como mínimo de 100 €). En este caso, hay un conjunto de instrucciones, las que aplican el descuento, que sólo se ejecutan cuando se cumple la condición. En caso contrario, se ignoran y el precio final es el mismo que el original.

La estructura de selección simple permite controlar el hecho de que se ejecute un conjunto de instrucciones si y sólo si se cumple la condición lógica (es decir, el resultado de evaluar la condición lógica es igual a true). En caso contrario, no se ejecutan.

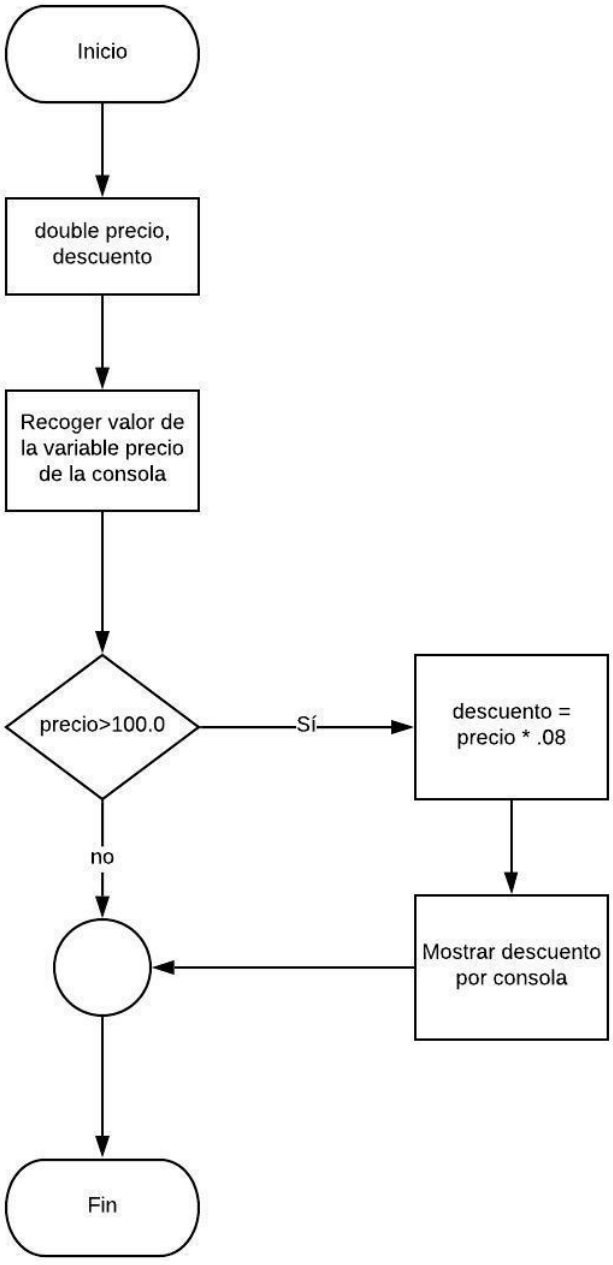
Para llevar a cabo este tipo de control sobre las instrucciones del programa, hay que usar una sentencia **if** (si ...). En el caso del Java, la sintaxis es la siguiente:

```
instrucciones del programa

if ( expresión booleana ) {
      Instrucciones para ejecutar si la expresión evalúa a true - Bloque if
}

resto de instrucciones del programa
```

En el siguiente ejemplo el programa pide al usuario el precio de un artículo, si el precio es mayor que 100 euros entonces se aplica un descuento del 8% y se muestra el valor del descuento por consola.

```
package dam.programacion;

import java.util.Scanner;

public class Descuento {
	//Precio mínimo al que se aplicará descuento
	final static double PRECIO_MIN = 100.0;
	//Constante para porcentaje del descuento: 8%
	final static double DESCUENTO = .08;

	public static void main(String[] args) {


		//Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		//Mostrar información al usuario
		System.out.println("Introduce el precio del artículo");


		//Leer valor
		double precio = sc.nextDouble();
		sc.nextLine();

		//Calcular descuento
		double descuento = 0.0;
		//Estructura de selección simple
		if(precio>PRECIO_MIN)
		{
			descuento = precio * DESCUENTO;
			System.out.print("Este producto tiene un descuento de "
+ descuento + " euros");
		}


		//Cerrar el objeto scanner
		sc.close();
	}

}
```

Veamos el flujo de ejecución del programa:



La introducción de estructuras de selección en un programa complica la sintaxis del código fuente, por lo que hay que ser muy cuidadoso a la hora de escribir una sentencia ***if***. Hay que prestar atención a los siguientes aspectos:

- La expresión booleana que denota la condición lógica puede ser tan compleja como se quiera, pero debe estar siempre entre paréntesis.
- Las instrucciones a ejecutar si la condición es cierta están englobadas entre dos llaves (**{ }**). Este conjunto se considera un bloque de instrucciones asociado a la sentencia ***if*** (*bloqueif*).

- La línea donde están las llaves o la condición no acaba nunca en punto y coma (;), al contrario que otras instrucciones.
- Aunque no es imprescindible, es una buena costumbre que las instrucciones del bloque estén identadas.

Visto esto, vale la pena destacar que con la introducción de estructuras de selección simple también aparece por primera vez código fuente con diferentes bloques de instrucciones: el del método principal y los asociados a la sentencia ***if***. La principal característica de este hecho es que la relación entre bloques es jerárquica: todos los nuevos bloques de instrucciones son subbloques del método principal.

Como iremos viendo, esta circunstancia se repetirá en todas las estructuras de control de un programa. Por eso debe tener muy claro donde empieza y termina cada bloque, y qué bloques son subbloques de otro.

También hay que hacer hincapié en que, cuando se usen estructuras que delimitan bloques de instrucciones diferentes, identificados por estar escritos entre llaves (***{ }***), es importante identar cada línea. De este modo, se facilita la legibilidad del código fuente y la identificación de cada bloque de instrucciones. Si nos fijamos, vermos que, de hecho, esto ya se había aplicado hasta ahora a las instrucciones dentro del método principal o *main*, también delimitadas por llaves, respecto a la declaración de inicio de la clase. Todas las instrucciones del método principal están identadas respecto al margen de la declaración del método principal.

Errores típicos sentencia ***if***.

Mensaje de error al compilar	Error cometido	Ejemplo
'(' Expected	Expresión no rodeada de paréntesis	if precio > PRECIO_MIN) {
';' expected	Falta ; en alguna instrucción del bloque if	descuento = precio * DESCUENTO
Siempre ejecuta bloque if	Se ha puesto ; a la sentencia if	if (precio > PRECIO_MIN); {
Sólo hace condicionalmente la primera instrucción del bloque if, el resto las hace siempre	Faltan las claves que han de rodear el bloque, {…}	if (precio > PRECIO _MIN)

Vinculado al último error, cabe mencionar que cuando el bloque *if* sólo tiene una única instrucción, el uso de llaves es opcional. De todos modos, es muy recomendable usarlas siempre, independientemente de este hecho. Esto facilita la identificación del bloque de código asociado a la sentencia *if* cuando se está leyendo el código fuente. Además, esto también facilitará el mantenimiento; un error típico es dejar un *if* sin llaves porque sólo se realiza una sentencia dentro del *if*, pero con el tiempo necesitamos añadir más sentencias al *if*, las escribimos debajo de la existente y olvidamos añadir las llaves que ahora sí son necesarias. Es siempre una buena práctica incluir las llaves a los bloques *if*.

1.2 Selección doble (if / else)

La estructura de **selección doble** permite controlar el hecho de que se ejecute un conjunto de instrucciones, sólo si se cumple la condición lógica, y que se ejecute otro, sólo si no se cumple la condición lógica.

En este caso, y a diferencia de la anterior, ahora hay dos escenarios excluyentes. Nunca puede pasar que ambos bloques se acaben ejecutando.

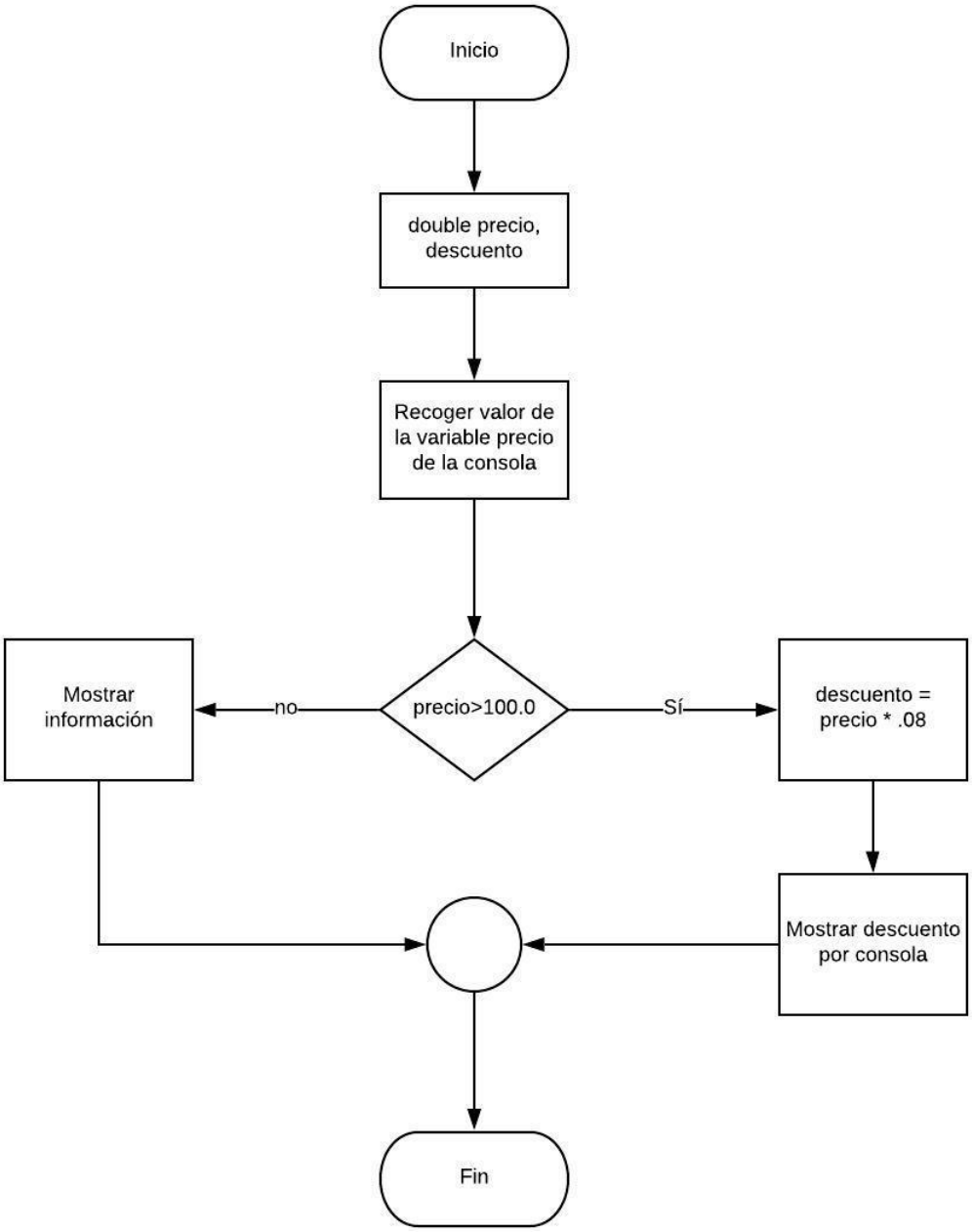
Para llevar a cabo este tipo de control sobre las instrucciones del programa, hay que usar una sentencia **if/else** (*si ... si no ...*). En el lenguaje Java, la sintaxis es la siguiente:

```
instrucciones del programa

if (expresión booleana) {
      Instrucciones a ejecutar si la expresión evalúa a true - Bloque if
} else {
      Instrucciones a ejecutar si la expresión evalúa a false - Bloque else
}

resto de instrucciones del programa
```

Retomando el ejemplo anterior, cuando el producto no tenga descuento se mostrará esta información por pantalla

```
import java.util.Scanner;

public class Descuento {
	//Precio mínimo al que se aplicará descuento
	final static double PRECIO_MIN = 100.0;
	//Constante para porcentaje del descuento: 8%
	final static double DESCUENTO = .08;

	public static void main(String[] args) {


		//Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		//Mostrar información al usuario
		System.out.println("Introduce el precio del artículo");


		//Leer valor
		double precio = sc.nextDouble();
		sc.nextLine();

		//Calcular descuento
		double descuento = 0.0;
		//Estructura de selección simple
		if(precio>PRECIO_MIN)
		{
			descuento = precio * DESCUENTO;
			System.out.print("Este producto tiene un descuento de "
+ descuento + " euros");
		}
		else
		{
			System.out.print("Este producto no tiene descuento");
		}


		//Cerrar el objeto scanner
		sc.close();
	}

}
```

El diagrama de flujo correspondiente es el siguiente:



1.3 Selección múltiple (if / else if … / else)

Finalmente, a la hora de establecer el flujo de control de un programa, también existe la posibilidad de que haya un número arbitrario de caminos alternativos, no sólo dos. Por ejemplo, imaginemos un programa que, a partir de la nota numérica de un examen, establece cuál es la calificación del alumno. Por eso habrá que ver dentro de qué rango se encuentra el número. En cualquier caso, los resultados posibles son más de dos.

La estructura de **selección múltiple** permite controlar el hecho de que al cumplirse un caso entre un conjunto finito de casos se ejecute el conjunto de instrucciones correspondiente.

La sintaxis es la siguiente

```
instrucciones del programa

if (expresión booleana 1) {
      Instrucciones a ejecutar si la expresión 1 evalúa a true (cierto) - 
Bloque if
} else if (expresión booleana 2) {
      Instrucciones a ejecutar si la expresión 2 evalúa a true (cierto) - 
Bloque if

... se repite tanto veces como sea necesario ...

} else if (expresión booleana N) {
      Instrucciones a ejecutar si la expresión N evalúa a true (cierto) - 
Bloque if
} else {
      Instrucciones a ejecutar si todas las expresiones evalúan a false- 
Bloque else
}

resto de instrucciones del programa
```

Este caso es un poco más complicado, ya que hay más de una condición lógica dentro de la sentencia. Cada condición se evaluando por orden, de arriba a abajo, y se ejecutará el código del primer caso en el que la expresión evalúe como cierto. Si una de las condiciones es cierta, ya no se vuelve a evaluar ninguna de las condiciones restantes y se ignora el resto de bloques. Si en este proceso se da el caso de que ninguna de las expresiones es verdadera, se ejecutarán las instrucciones del bloque *else*.

El punto importante de esta sentencia es que sólo se ejecutará un único bloque de todos los posibles. Incluso en el caso de que más de una de las expresiones booleanas pueda evaluar cierto, sólo se ejecutará el bloque asociado a la primera de estas expresiones dentro del orden establecido en la sentencia.

También es destacable el hecho de que el bloque *else* es opcional. Si no queremos, no hay que ponerlo. Si ese fuera el caso y no se cumpliera ninguna de las condiciones, no se ejecutaría ninguno de los bloques.

En el siguiente ejemplo dada una nota numérica se obtiene una calificación (texto): Matrícula, Sobresaliente, Notable, etc. de acuerdo con dicha nota numérica.

```
import java.util.Scanner;

public class Notas {

	//Constantes para los valores mínimos de las notas
	static final double MATRICULA = 10.0;
	static final double SOBRESALIENTE = 9.0;
	static final double NOTABLE = 7.0;
	static final double BIEN = 6.0;
	static final double APROBADO = 5.0;

	public static void main(String[] args) {

		//Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		//Mostrar información al usuario
		System.out.println("Introduce la nota numérica");

		//Leer valor
		double nota = sc.nextDouble();
		sc.nextLine();

		//Obtener calificación
		String calificacion = "";
		if(nota>=MATRICULA) {
			calificacion = "Matrícula de honor";
		} else if(nota>=SOBRESALIENTE) {
			calificacion = "Sobresaliente";
		} else if(nota>=NOTABLE) {
			calificacion = "Notable";
		} else if(nota>=BIEN) {
			calificacion = "Bien";
		} else if(nota>=APROBADO) {
			calificacion = "Aprobado";
		} else {
			calificacion = "Suspenso";
		}

		//Mostrar calificación por consola
		System.out.println("Tu calificación es " + calificacion);

		//Cerrar Scanner de la consola
		sc.close();
	}

}
```

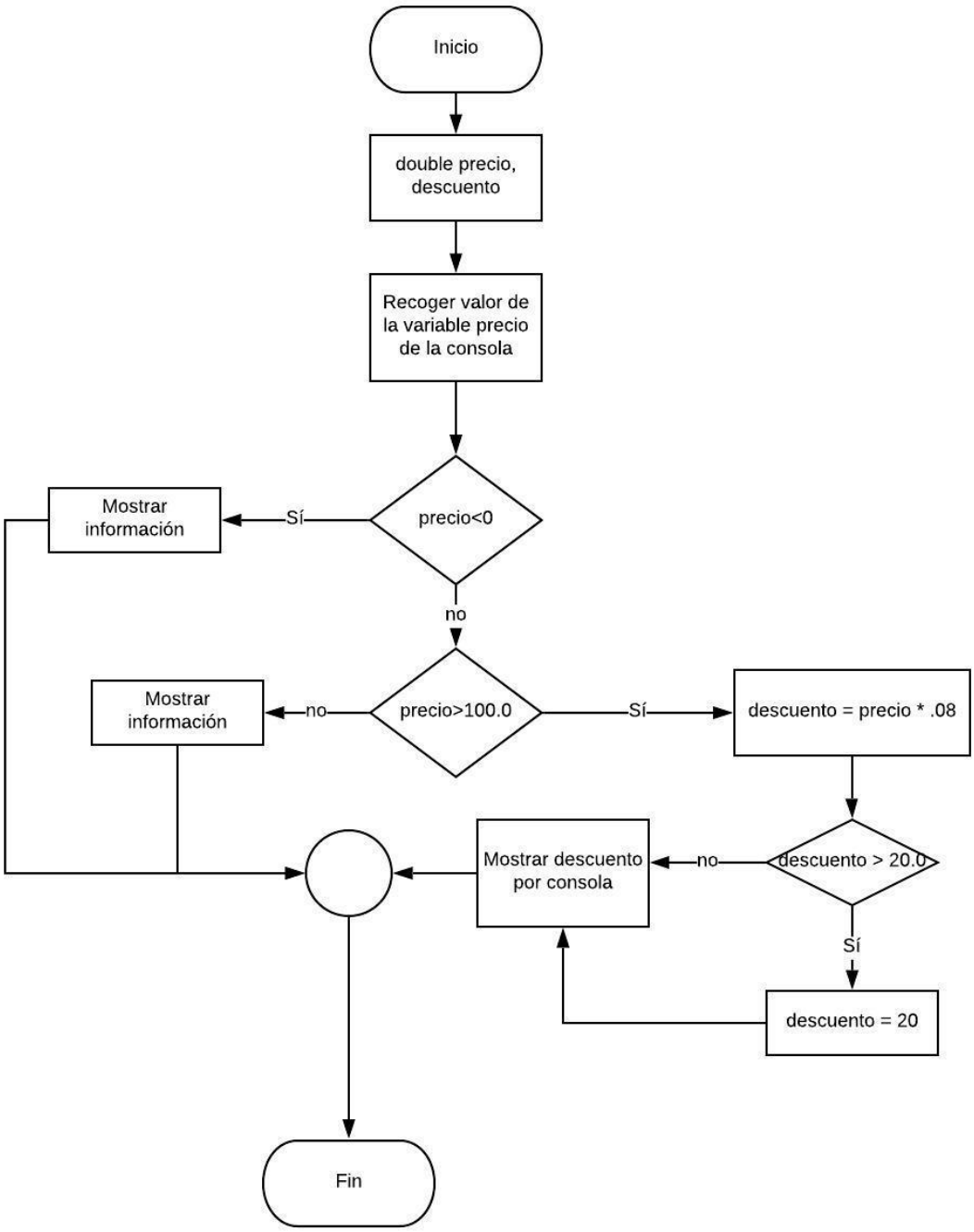
1.3.1 Anidamiento de estructuras de selección

Las sentencias que definen estructuras de selección son instrucciones como cualquier otra dentro de un programa, si bien con una sintaxis un poco más complicada. Por lo tanto, nada impide que vuelvan a aparecer dentro de bloques de instrucciones de otras estructuras de selección. Esto permite crear una disposición de bifurcaciones dentro del flujo de control a fin de dotar al programa de un comportamiento complejo, para comprobar si se cumplen diferentes condiciones de acuerdo con cada situación.

Siguiendo con el ejemplo del descuento podemos incluir un descuento máximo de 20 euros y una validación de precio negativo para informar al usuario de esta circunstancia

```
import java.util.Scanner;

public class Descuento {
	//Precio mínimo al que se aplicará descuento
	static final double PRECIO_MIN = 100.0;
	//Constante para porcentaje del descuento: 8%
	static final double DESCUENTO = .08;
	//Descuento máximo aplicado
	static final double DESCUENTO_MAX = 20.0;

	public static void main(String[] args) {


		//Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		//Mostrar información al usuario
		System.out.println("Introduce el precio del artículo");


		//Leer valor
		double precio = sc.nextDouble();
		sc.nextLine();

		//Calcular descuento
		double descuento = 0.0;
		//Estructura de selección múltiple
		if(precio<0)
		{
			System.out.print("El precio no puede ser negativo");
		}
		else if(precio>PRECIO_MIN)
		{
			descuento = precio * DESCUENTO;
			//Estructura de selección anidada
			//Si el descuento es mayor que 20 dejarlo en 20
			if(descuento > DESCUENTO_MAX)
			{
				descuento = DESCUENTO_MAX;
			}
			System.out.print("Este producto tiene un descuento de "
+ descuento + " euros");
		}
		else
		{
			System.out.print("Este producto no tiene descuento");
		}


		//Cerrar el objeto scanner
		sc.close();
	}

}
```

Veamos cómo queda el diagrama de flujo:



1.3.2 Ámbito de variables

En el momento que el código fuente de un programa se organiza en diferentes bloques de instrucciones, unos dentro de otros, debido a la aparición de estructuras de control, hay que tener mucho cuidado de respetar el ámbito de la declaración de una variable.

Debemos recordar que la declaración de una variable sólo tiene validez desde que se declara hasta el delimitador de final del bloque donde se ha declarado (la llave **}**).

Si declaramos una variable dentro de un bloque *if* o *else* no podremos usarla fuera de ese bloque.

Como regla general deberíamos declarar las variables de manera que su ámbito sea lo más reducido posible.

Declarar las variables justo antes del momento de usarlas facilita la lectura del mismo al no tener que verificar el valor anterior de una variable. Además.

1.4 El conmutador (switch)

Hay una estructura de selección un poco especial, por lo que se ha dejado para el final. Lo que la hace especial es que no se basa en evaluar una condición lógica compuesta por una expresión booleana, sino que establece el flujo de control a partir de la evaluación de una expresión de tipo byte, short, int, char o String (también para tipos Enum, todavía no hemos visto este tipo de objetos).

La sentencia **switch** enumera, uno por uno, un conjunto de valores discretos que se quieren tratar, y asigna las instrucciones a ejecutar a cada valor diferente de la expresión. Finalmente, especifica qué hacer si el valor de la expresión no corresponde a ninguno de los valores enumerados. Es como un conmutador en el que se asigna un código para cada valor posible tratar. Este comportamiento sería equivalente a una estructura de selección múltiple en el que, implícitamente, todas las condiciones son comparar si una expresión es igual a cierto valor. Cada rama controlaría un valor diferente. El caso final es equivalente al *else*.

De hecho, en la mayoría de casos, esta sentencia no aporta nada diferente a una estructura de selección múltiple desde el punto de vista del flujo de control. Pero es muy útil de cara a mejorar la legibilidad del código o facilitar la generación del código del programa. Ahora bien, sí hay un pequeño detalle en que esta sentencia aporta algo que el resto de estructuras de selección no pueden hacer directamente: ejecutar de manera consecutiva más de un bloque de código relativo a diferentes condiciones.

La sintaxis de la sentencia ***switch*** es la siguiente:

```
instrucciones del programa
switch (expresión de tipo entero) {
      case valor1:
            instrucciones si la expresión vale valor1
            (Opcionalmente) break;
      case valor2:
            instrucciones si la expresión vale valor2
            (Opcionalmente) break;
      ...
      case valorN:
            instrucciones si la expresión vale valorN
            (Opcionalmente) break;
      default:
            instrucciones si la expresión no es ninguno de los valores
anteriores
}
```

```
resto de instrucciones del programa
```

De nuevo, se basa en una palabra clave seguida de una condición entre paréntesis y un bloque de código entre llaves. Ahora bien, la diferencia respecto de las sentencias vistas hasta ahora es que los conjuntos de instrucciones asignados a cada caso independiente no se distinguen entre sí por claves. Estos se enumerando como un conjunto de apartados etiquetados por la palabra clave case, seguida del valor que se quiere tratar y de dos puntos. De manera opcional, se puede poner una instrucción especial que sirve de delimitador de final de apartado, llamada break. Al final de todos los apartados se pone uno especial, llamado *default*.

Al ejecutar la sentencia *switch* se evalúa la expresión entre paréntesis e inmediatamente se ejecuta el conjunto de instrucciones asignado a ese valor entre los diferentes apartados *case*. Si no hay ninguno con este valor entre los disponibles, entonces se ejecuta el apartado etiquetado como *default* (por defecto).

Hay que decir que los valores numéricos no deben seguir ningún orden en concreto para definir cada apartado. Si bien puede ser más limpio enumerarlos por orden creciente, esto no tiene ningún efecto sobre el comportamiento del programa.

La particularidad especial de esta sentencia la tenemos en el uso de la instrucción break. Esta indica qué hay que hacer una vez ejecutadas las instrucciones del apartado. Si aparece la instrucción, el programa se salta el resto de apartados y continúa con la instrucción posterior a la sentencia *switch* (después de la llave que cierra el bloque). En este aspecto, si todos los apartados terminan en *break*, el funcionamiento de *switch* es equivalente a hacer una selección múltiple con *if / else if / …/ else*.

Si bien el uso de la palabra *break* es opcional y nos permitiría seguir ejecutando instrucciones de la siguiente etiqueta se recomienda de manera general su uso después antes de la etiqueta siguiente para mejorar la legibilidad del código y evitar errores de flujo en los programas.

En el siguiente ejemplo se implementa una calculadora simple, primero se pide al usuario que introduzca 2 números reales como operandos, después se ofrece un menú de operaciones 1 – suma, 2 -resta, 3 – producto, 4 – cociente, a continuación, el usuario debe introducir la opción deseada y después se muestra el resultado de la operación por consola. Para calcular la operación en función de la opción seleccionada se ha utilizado una instrucción *switch*.

```java
import java.util.Scanner;

public class Calculadora {

	public static void main(String[] args) {
		//Inicializar objeto Scanner para leer de la consola
	    Scanner sc = new Scanner(System.in);

	    //Mostrar información al usuario y recoger valores desde el teclado
	    System.out.println("Calculadora");
	    System.out.println("Introduce el primer operando");
	    double a = sc.nextDouble();
	    sc.nextLine();
	    System.out.println("Introduce el segundo operando");
	    double b = sc.nextDouble();
	    sc.nextLine();
	    System.out.println();
	    System.out.println("1 - Suma");
	    System.out.println("2 - Resta");
	    System.out.println("3 - Producto");
	    System.out.println("4 - Cociente");
	    System.out.println("Elige una opción");
	    int opcion = sc.nextInt();
	    sc.nextLine();

	    //Variables para el resultado de la operación y la operación
elegida
	    double resultado = 0.0;
	    String operacion = "";
	    //Selección múltiple aplicada con switch
	    switch (opcion) {
	      case 1:
	    		operacion = " + ";
	    		resultado = a + b;
	    		break;
	      case 2:
	    		operacion = " - ";
	    		resultado = a - b;
	    		break;
	      case 3:
	    		operacion = " * ";
	    		resultado = a * b;
	    		break;
	      case 4:
	    		operacion = " / ";
	    		resultado = a / b;
	    		break;

	      default:
	    		operacion = "";
	    		break;
	      }

	    //Mostrar el resultado
	    if(operacion.equals("")) {
	      System.out.println("No se ha seleccionado una operación
válida");
	    } else {
	      System.out.println(a + operacion + b + " = " + resultado);
	    }

	      //Cerrar Scanner de la consola
	      sc.close();

	}

}
```

Veamos ahora un ejemplo de instrucción *switch* en la que se hace uso de la posibilidad de no incluir *breaks* en algunas de las opciones. Se trata de un programa que devuelve el número de días del mes.

```
import java.util.Scanner;

public class DiasMes {
	public static void main(String[] args) {
		// Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		// Mostrar información al usuario y recoger valores desde el teclado
		System.out.println("Días del mes");
		System.out.println("Introduce número de mes");
		int mes = sc.nextInt();
		sc.nextLine();

		// Selección múltiple aplicada con switch aplicando
		// las mismas instrucciones para varias opciones
		int numeroDias = 0;
		switch (mes) {
		//Para todos estos casos el número de días es 31
		case 1:
		case 3:
		case 5:
		case 7:
		case 8:
		case 10:
		case 12:
			numeroDias = 31;
			break;
		case 2:
			numeroDias = 28;
			break;
		//Para todos estos meses el número de días es 30
		case 4:
		case 6:
		case 9:
		case 11:
			numeroDias = 30;
			break;
		}

		// Mostrar el resultado
		if (numeroDias == 0) {
			System.out.println("Mes no válido, debe estar entre 1 y 12");
		} else {
			System.out.println("Días del mes: " + numeroDias);
		}

		// Cerrar Scanner de la consola
		sc.close();
	}
}
```

1.5 Validación de datos de entrada con estructuras de selección

Las estructuras de selección son especialmente útiles como mecanismo para controlar si los datos que ha introducido un usuario son correctos antes de usarlas dentro del programa. Para controlar si el valor que se ha introducido desde la consola corresponde con el tipo de dato esperado para cada instrucción de lectura tenemos disponibles los siguientes métodos en la case Scanner:

Método	Tipo de dato controlado
hasNextByte()	byte
hasNextShort()	short
hasNextInt()	int
hasNextLong()	long
hasNextFloat()	float
hasNextDouble()	double
hasNextBoolean()	boolean

No existe un método asociado a una cadena caracteres, ya que cualquier dato escrito por el teclado siempre se puede interpretar como texto y la lectura como tal nunca puede dar error.

En el siguiente ejemplo vemos cómo podemos hacer uso de este mecanismo para verificar que el dato introducido es válido y así evitar un error en tiempo de ejecución:

```java
import java.util.Scanner;

public class DiasMes {
	public static void main(String[] args) {
		// Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		// Mostrar información al usuario y recoger valores desde el teclado
		System.out.println("Días del mes");
		System.out.println("Introduce número de mes");
		//Validar que el valor introducido es un número entero
		//De esta manera se evita un error de ejecución si el formato no es válido
		if (sc.hasNextInt()) {
			int mes = sc.nextInt();
			sc.nextLine();

			// Selección múltiple aplicada con switch
			// aplicando las mismas instrucciones para varias opciones
			int numeroDias = 0;
			switch (mes) {
			// Para todos estos casos el número de días es 31
			case 1:
			case 3:
			case 5:
			case 7:
			case 8:
			case 10:
			case 12:
				numeroDias = 31;
				break;
			case 2:
				numeroDias = 28;
				break;
			// Para todos estos meses el número de días es 30
			case 4:
			case 6:
			case 9:
			case 11:
				numeroDias = 30;
				break;
			}

			// Mostrar el resultado
			if (numeroDias == 0) {
				System.out.println("Mes no válido, debe estar entre 1 y 12");
			} else {
				System.out.println("Días del mes: " + numeroDias);
			}
		} else {
			System.out.println("Debe introducir un valor numérico entero para el mes");
		}
		// Cerrar Scanner de la consola
		sc.close();
	}
}
```

2 Estructuras iterativas

Las **estructuras de repetición** o **iterativas** permiten repetir una misma secuencia de instrucciones varias veces, mientras se cumpla una cierta condición.

En su aspecto general, se parecen a las estructuras de selección. Hay una sentencia especial que hay que escribir el código fuente, unida a una condición lógica y un bloque de código (en este caso, siempre será sólo uno). Pero en este caso, mientras la condición lógica sea cierta, toda la secuencia de instrucciones se ejecutando repetidamente. En el momento en que se deja de cumplir la condición, se deja de ejecutar el bloque de código y ya se sigue con la instrucción que hay después de la sentencia de la estructura de repetición.

Llamamos **bucle** al conjunto de instrucciones que se debe repetir un cierto número de veces, y llamamos **iteración** cada ejecución individual del bucle.

Como ocurre con las estructuras de selección, hay diferentes tipos de sentencias, cada una con sus particularidades. Normalmente, la diferencia principal está vinculada al momento en que se evalúa la condición para ver si hay que volver a repetir el bloque de instrucciones o no. A lo largo de este apartado, las veremos con detalle.

2.1 Control de las estructuras iterativas

Como ocurría con las estructuras de selección, las estructuras de repetición no tienen sentido si no es que la condición lógica depende de alguna variable que pueda ver modificado su valor para diferentes ejecuciones. En caso contrario, la condición siempre valdrá lo mismo para cualquier ejecución posible y usar una estructura de control no será muy útil. En este caso, si la condición siempre es false, nunca se ejecuta el bucle, por lo que es código inútil. Pero, para las estructuras de repetición, si la condición siempre es true el problema es mucho más grave. Como absolutamente siempre que se evalúa si es precisa una nueva iteración, la condición se cumple, el bucle no se deja nunca de repetir. ¡El programa nunca acabará!

Un **bucle infinito** es una secuencia de instrucciones dentro de un programa que itera indefinidamente, normalmente porque se espera que se alcance una condición que nunca se llega a producir.

Un bucle infinito es un error semántico de programación. Si los hay, el programa compilará perfectamente de todos modos.

Cuando esto sucede, el programa no se puede parar de otra manera que no sea cerrando directamente desde el sistema operativo (por ejemplo, cerrando la ventana asociada o con alguna secuencia especial de escape del teclado) o usando algún mecanismo de control de la ejecución de su programa que ofrezca el IDE usado.

Teniendo en cuenta el peligro que un programa acabe ejecutando indefinidamente, forzosamente dentro de todo bucle debe haber instrucciones que manipulen variables el valor de las que permita controlar la repetición o el final del bucle. Estas variables se llaman **variables de control**.

Garantizar la asignación correcta de valores de las variables de control de una estructura repetitiva es extremadamente importante. Es imprescindible que el código permita que, en algún momento, la variable cambie de valor, por lo que la condición lógica se deje de cumplir. Si esto no es así, tendrá un bucle infinito.

Normalmente, las variables de control dentro de un bucle se pueden englobar dentro de alguno de estos tipos de comportamiento:

- **Contador:** una variable de tipo entero que va aumentando o disminuyendo, indicando de manera clara el número de iteraciones que habrá que hacer.
- **Acumulador:** una variable en la que se van acumulando directamente los cálculos que se quieren hacer, de manera que al alcanzar cierto valor se considera que ya no es necesario hacer más iteraciones. Si bien se parecen a los contadores, no son exactamente lo mismo.
- **Semáforo:** una variable que sirve como interruptor explícito de si hay que seguir haciendo iteraciones. Cuando ya no queremos hacer más, el código simplemente se encarga de asignarle el valor específico que servirá para que la condición evalúe false. Los semáforos también se llaman habitualmente *flags*.

Evidentemente, una condición lógica puede tomar la forma de una expresión muy compleja, pero para empezar basta de conocer estos tres modelos. A veces, las diferencias pueden ser sutiles, y por eso tampoco hay que preocuparse demasiado si no se tiene claro a qué tipo pertenece exactamente la variable de control. Igualmente, tener presentes estos tres papeles básicos puede ser de ayuda en orden a enfocar la programación de una estructura de selección.

2.2 Sentencia while

La estructura de repetición por antonomasia es la codificada mediante la sentencia ***while***. Esta existe de una manera u otra en la mayoría de lenguajes de programación. Su particularidad es que prácticamente cualquier código basado en una estructura de repetición se puede representar usando esta sentencia. Con esta ya tendríamos suficiente para tratar casi cualquier caso posible. Esto sucede con contraposición de las estructuras de selección, en el que cada tipo de sentencia ofrece diferentes posibilidades.

La sentencia ***while*** permite repetir la ejecución del bucle mientras se verifique la condición lógica. Esta condición se verifica al principio de cada iteración. Si la primera vez, justo cuando se ejecuta la sentencia por primera vez, ya no se cumple, no se ejecuta ninguna iteración.

Su sintaxis en Java es la siguiente:

```
instrucciones del programa

while (expresión_booleana) {
      instrucciones para ejecutar dentro del bucle
}

resto de instrucciones del programa
```

Básicamente significa "*Mientras esto es cumpla, ejecuta esto otro*".

Su formato es muy similar a la sentencia if, simplemente cambiando la palabra clave para while. Como ya ocurría con las diferentes sentencias dentro de las estructuras de selección, si entre los paréntesis se pone una expresión que no evalúa un resultado de tipo booleano, habrá un error de compilación.

Ejemplos

1. Contador de 1 a 100

```
public class Contador {

      public static void main(String[] args) {
            //Declaración de la variable contador
            int contador = 0;
            //Bucle while, inclue la condición de repetición
            while(contador <100) {

                  //Mostramos el número por consola
                  //Se aplica primero el incremento para mostrar de 1 a 100
                  //Es importante no olvidar incrementar el valor del contador
                  //para evitar un bucle infinito
                  System.out.println(++contador);
            }

      }

}
```

Tabla valores del bucle

Iteración	Valor de i al inicio	Condición	Valor de i al final
1	0	O<100, true	1
2	1	1<100, true	2
3	2	2<100, true	3
…			
100	99	99<100, true	100
101	100	100<100, false	Ya ha salido del bucle

2. Tabla de multiplicar de un número

Tabla valores del bucle, por ejemplo, para un valor de entrada por consola de 5

Iteración	Valor de **i** al inicio	**Condición**	Valor de **resultado**	Valor de **i** al final
1	1	1<=10, true	5	1
2	2	2<=10, true	10	2
3	3	3<=10, true	15	3
…				
10	9	10<=10, true	50	10
11	11	11<=10, false	- (ya ha salido)	- (ya ha salido)

Vamos con el programa:

```
public class TablaMultiplicar {

	public static void main(String[] args) {

		//Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		//Mostrar información al usuario
		System.out.println("Tabla de multiplicar");
		System.out.println("Introduce un número");

		//Leer valor, nos aseguramos de que es un entero antes de seguir
		if(sc.hasNextInt()) {
			int numero = sc.nextInt();
			sc.nextLine();

			//Contador i, irá de 1 a 10
			int i=1;
			while(i<=10) { //Condición del bucle
				System.out.println(i + " x " + numero + " = " + i*numero);
				i++; //incremento, no olvidar para evitar bucle infinito
			}

		} else {
			System.out.println("El número debe ser un entero");
		}
		// cerrar scanner
		sc.close();
	}

}
```

3. Calcular el resto o módulo de una división entre enteros

El módulo calcula el resto de dividir un entero (el dividendo) por otro (el divisor). Una estrategia simple para calcularlo es ir restando el divisor al dividendo hasta que ya no se puede hacer más, ya que daría negativo. En este caso, el valor del dividendo se va modificando directamente hasta encontrar la solución.

Si se estructura paso a paso, sería:

1. Se pregunta cuál es el dividendo.
2. Se lee.
3. Se pregunta cuál es el divisor.
4. Se lee.
5. Si el dividendo es más pequeño que el divisor, ya hemos terminado. El divisor ya es resultado del módulo.
6. En caso contrario, el dividendo se le resta el divisor.
7. Se vuelve a comprobar el paso 5.

El código quedaría de la siguiente manera

```java
import java.util.Scanner;

public class Modulo {

	public static void main(String[] args) {
		// Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		// Mostrar información al usuario
		System.out.println("Cálcula el resto de la división");

		// Leer valor dividendo
		System.out.println("Introduce el dividendo");
		// nos aseguramos de que es un entero antes de seguir
		if (sc.hasNextInt()) {
			int dividendo = sc.nextInt();
			sc.nextLine();

			// Leer valor divisor
			System.out.println("Introduce el divisor");
			// nos aseguramos de que es un entero antes de seguir
			if (sc.hasNextInt()) {
				int divisor = sc.nextInt();
				sc.nextLine();

				//Variable para el mensaje de salida
				String mensaje = dividendo + " % " + divisor + " = ";

				// Cálculo del resto
				// se va restando en bucle el divisor al dividendo
				// mientras el dividendo sea mayor que el divisor
				// cuando termina el resultado es el resto
				while(dividendo > divisor) {
					dividendo-=divisor;
				}
				//Mostrar el resultado
				mensaje = mensaje + dividendo;
				System.out.println(mensaje);

			} else {
				System.out.println("El divisor debe ser un entero");
			}

		} else {
			System.out.println("El dividendo debe ser un entero");
		}

		// Cerrar el scanner
		sc.close();
	}

}
```

Puedes practicar a generar la tabla de valores del bucle para valores de ejemplo de dividendo y divisor.

Iteración	Valor de **dividendo** al inicio	**Condición** dividendo > divisor	Valor de **dividendo** al final

4. Uso de semáforo o *flag* para salir del bucle. Programa adivina un número

Vamos a estudiar es el uso de un semáforo para indicar de forma explícita si ya no hay que hacer iteraciones. Esta estrategia de uso de variables de control se basa en situaciones en las que decidir si se quiere continuar haciendo iteraciones de un bucle no se puede predecir o calcular de acuerdo con un valor que va aumentando o disminuyendo. Simplemente, hay que ir repitiendo hasta que se cumpla una condición muy concreta. Entonces, de lo que se dispone es de una variable de control, normalmente de tipo booleano, sobre la que se hace una asignación explícita que provocará que la condición lógica sea false y se salga del bucle.

Un ejemplo de este caso es un programa en el que hay que adivinar un valor secreto. Mediante una estructura de selección es posible establecer si se ha acertado o no, pero lo que no tiene sentido es que cada vez que se quiera intentar adivinarlo, se haya de ejecutar el programa de nuevo. Lo más normal es que se pregunte al usuario hasta que acierte. En este caso, sin embargo, no hay un valor que poco a poco, gradualmente, va variando hasta poder establecer que hay que dejar de iterar. Se pasa de golpe de continuar preguntando a que ya no sea necesario, según la condición de si se ha acertado o no. Esta se puede dar en cualquier iteración, pero es imposible estimar cuando, ya que depende totalmente del valor introducido por el usuario.

Si describimos lo que hay que hacer paso por paso, sería:

1. Decidir cuál será el número para adivinar.
2. Pedir que se introduzca un número por el teclado.
3. Leerlo.
4. Si el número no es el valor secreto:
    1. Hay que avisar que se ha fallado.
    2. Volver al paso 2.
5. De lo contrario, ya hemos terminado. Mostrar una felicitación.

```java
import java.util.Scanner;

public class AdivinaNumero {

	public static void main(String[] args) {
		// Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		// Mostrar información al usuario
		System.out.println("Adivina el número");

		// selección de número a adivinar
		int secreto = 4;

		// variable flag o semáforo del bucle
		boolean acierto = false;

		//Condicion de iteración del bucle
		while(!acierto) {

			// leer numero
			System.out.println("Introduce un número");
			int numero = sc.nextInt();

			// actualizar flag
			acierto = (numero==secreto);

		}

		// mensaje de acierto
		System.out.println("¡Enhorabuena! Has ganado el número es " + secreto);

		// Cerrar el scanner
		sc.close();
	}

}
```

5. Combinar el uso de flag y contador para salir del bucle.

Si en el programa anterior queremos limitar el número de intentos podemos añadir un contador y evaluar ambas variables. Veamos cómo quedaría:

```
import java.util.Scanner;

public class AdivinaNumero {

	static final int MAX_INTENTOS = 5;

	public static void main(String[] args) {
		// Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		// Mostrar información al usuario
		System.out.println("Adivina el número");

		// numero de intentos
		int intentos = 0;

		// selección de número a adivinar
		int secreto = 10;

		// variable flag o semáforo del bucle
		boolean acierto = false;

		//Condicion de acierto y número de intentos
		while(!acierto && intentos < MAX_INTENTOS) {

			// leer numero
			System.out.println("Introduce un número");
			int numero = sc.nextInt();

			// actualizar flag
			acierto = (numero==secreto);

			// actualizar numero de intentos
			intentos++;
		}

		// verificar si se ha acertado y mostrar mensaje
		if(acierto) {
		System.out.println("Has ganado el número es " + secreto);
		} else {
			System.out.println("Superado el número de intentos");
			System.out.println("El número secreto era " + secreto);
		}

		// Cerrar el scanner
		sc.close();
	}

}
```

2.3 Sentencia do / while

La sentencia ***do / while*** permite repetir la ejecución del bucle mientras se verifique la condición lógica. A diferencia de la sentencia *while*, la condición se verifica al final de cada iteración. Por lo tanto, independientemente de cómo evalúe la condición, al menos siempre se llevará a cabo la primera iteración.

La sintaxis es la siguiente:

```
instrucciones del programa

do {
     instrucciones para ejecutar dentro del bucle
} while (expresión_booleana);

resto de instrucciones del programa
```

Es muy parecida a la sentencia *while*, pero invirtiendo el lugar donde aparece la condición lógica. Para poder distinguir claramente donde empieza el bucle se usa la palabra clave do.

Veamos el siguiente ejemplo en el que incluimos un bucle *do / while* para asegurar que se introduce correctamente un número entero en un determinado rango. Podríamos combinar este ejemplo con el segundo ejemplo del apartado anterior asegurando que las tablas de multiplicar se muestran para valores del 1 al 9.

```java
import java.util.Scanner;

public class EntradaNumero {

	public static void main(String[] args) {
		// Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		// flag numero correcto
		boolean correcto = false;
		int numero = 0;

		do {

			// Mostrar información al usuario
			System.out.println("Introduce un entero de 1 a 9");
			// leer numero
			numero = sc.nextInt();
			sc.nextLine();

			// verificar condicion
			if (numero >= 1 && numero <= 9) {
				correcto = true; // actualizar flag
			}

		} while (!correcto);

		// mensaje de fin
		System.out.println("El número introducido es " + numero);

		// Cerrar el scanner
		sc.close();

	}

}
```

2.4 Sentencia for

La sentencia ***for*** permite repetir un número determinado de veces un conjunto de instrucciones.

En algunas situaciones especiales ya se conoce *a priori* la cantidad exacta de veces que se repetirá el bucle. En tal caso es útil disponer de un mecanismo que represente de manera más clara la declaración de una variable de control de tipo contador, la especificación de hasta donde se debe contar, y que al final de cada iteración incremente o disminuya su valor de manera automática, en lugar de tener que hacerlo nosotros. Automatizar este último punto es muy importante, ya que evita que por un olvido no se haga y acabe generando un bucle infinito. Y, ojo, olvidarse de modificar una variable contador es un error muy típico cuando se usan estructuras de repetición.

La sintaxis es la siguiente:

```
instrucciones del programa

for (inicializar contador; expresión booleana; incremento contador) {
  instrucciones para ejecutar dentro del bucle
}

resto instrucciones del programa
```

La ejecución se realiza en el siguiente orden: inicialización del contador (sólo la primera vez), validación de la expresión booleana, bloque de instrucciones del bucle y finalmente el incremento contador para volver a iniciar la iteración con la validación de la expresión booleana y repetir el proceso mientras ésta se *true*.

A modo de ejemplo, podemos modificar el programa que calcula la tabla de multiplicar de un número usando un bucle *for*:

```
import java.util.Scanner;

public class TablaMultiplicar {

	public static void main(String[] args) {

		//Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		//Mostrar información al usuario
		System.out.println("Tabla de multiplicar");
		System.out.println("Introduce un número");

		//Leer valor, nos aseguramos de que es un entero antes de seguir
		if(sc.hasNextInt()) {
			int numero = sc.nextInt();
			sc.nextLine();

			// bucle for de 1 a 10
			for (int i = 1; i <= 10; i++) {
				// escribir tabla
				System.out.println(i + " x " + numero + " = " + i*numero);
			}

		} else {
			System.out.println("El número debe ser un entero");
		}
		//Cerrar el scanner
		sc.close();
	}
}
```

Como ocurría con las estructuras de selección, nada impide combinar diferentes estructuras de repetición, anidadas unas dentro de otras, para llevar a cabo tareas más complejas. En última instancia, la combinación y anidamiento de estructuras de selección y repetición conforman un programa.

Cuando se combinan estructuras de repetición hay que ser muy cuidadosos y tener presente qué variables de control están asociadas a la condición lógica de cada bucle. Tampoco hay que olvidar de identar correctamente cada bloque de instrucciones, para poder así identificar rápidamente donde termina y comienza cada estructura.

3 Sentencias de salto

Las sentencias de salto son aquellas que interrumpen de algún modo la ejecución de una sentencia de control.

3.1 Sentencia break

Esta sentencia interrumpe la ejecución de un bucle o de una sentencia *switch* (como ya hemos visto). La ejecución del programa continua en la línea siguiente al bucle o *switch*.

```
break;
```

Como ejemplo hemos modificado el programa “Adivina un Número” de forma que se ejecuta un break al adivinar al número para salir del bucle, de forma que el bucle únicamente tiene como condición el número de intentos.

```
import java.util.Scanner;

public class AdivinaNumero {

	static final int MAX_INTENTOS = 5;

	public static void main(String[] args) {
		// Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);
		// Mostrar información al usuario
		System.out.println("Adivina el número");
		// número de intentos
		int intentos = 0;
		// selección de número a adivinar
		int secreto = 10;
		// variable para guardar si hubo acierto
		boolean acierto = false;

		//Bucle para controlar el número de intentos
		while(intentos < MAX_INTENTOS) {

			// leer numero
			System.out.println("Introduce un número");
			int numero = sc.nextInt();

			// validar si se acierta el número
			if(numero==secreto) {
				//actualizar variable
				acierto = true;
				//salir del bucle
				break;
			}
			// actualizar numero de intentos
			intentos++;
		}
		// verificar si se ha acertado y mostrar mensaje
		if(acierto) {
		System.out.println("Has ganado el número es " + secreto);
		} else {
			System.out.println("Superado el número de intentos");
			System.out.println("El número secreto era " + secreto);
		}
		// Cerrar el scanner
		sc.close();
	}

}
```

3.2 Sentencia continue

Esta sentencia interrumpe la ejecución de una iteración del bucle y transfiere la ejecución a la condición del bucle para continuar si procede con la siguiente iteración.

```
continue;
```

En el siguiente ejemplo se utiliza la sentencia *continue* para no sumar el valor introducido al total cuando es cero y continuar con la ejecución de la siguiente iteración:

```
import java.util.Scanner;

public class CalcularMedia {

	public static void main(String[] args) {
		// Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);

		// Mostrar información
		System.out.println("Cálcular media aritmética");
		System.out.println("Introduce el número de valores a introducir");
		// introducción del número de valores asegurando el formato
		while (!sc.hasNextInt()) {
			sc.nextLine();
			System.out.println("Introduce el número de valores a introducir");
		}
		int contador = sc.nextInt(); // variable para el número de valores
		double total = 0.0; // variable que acumula la suma total de valores
		// Bucle para introducir valores
		for (int i = 0; i < contador; i++) {
			// introduccion de valor, asegurando el formato correcto
			System.out.println("Introduce el valor " + (i + 1));
			while (!sc.hasNextDouble()) {
				sc.nextLine();
				System.out.println("Introduce el valor \" + (i+1)");
			}
			double valor = sc.nextDouble();
			// si es cero no hace falta hacer la suma
			if (valor == 0.0)
				continue; // continua a la iteración siguiente
			// acumula el valor al total
			total += valor;
		}
		// calcular la media
		double media = total / contador;
		// mostrar resultado
		System.out.println("La media es " + media);
		// Cerrar el scanner
		sc.close();
	}

}
```

3.3 Sentencia return

Se utiliza *return* en un elemento del lenguaje que todavía no hemos estudiado en profundidad, las funciones o métodos. Se utiliza para finalizar la ejecución de la función y devolver el control a la línea que hacía la llamada a dicha función. Irá seguida del valor de retorno de la función (si lo tuviera).

```
return [valor de retorno de la función];
```

En el ejemplo se utiliza *return* para finalizar el programa en función de la opción de cálculo introducida por el usuario, cuando introduce 0 el programa termina al ejecutar un *return* en el método *main*:

```
import java.util.Scanner;
public class Calculadora {
	public static void main(String[] args) {
		// Inicializar objeto Scanner para leer de la consola
		Scanner sc = new Scanner(System.in);
		// Mostrar información al usuario y recoger valores desde el teclado
		System.out.println("Calculadora");
		System.out.println("1 - Suma");
		System.out.println("2 - Resta");
		System.out.println("3 - Producto");
		System.out.println("4 - Cociente");
		System.out.println("Elige una opción (0 para salir)");
		int opcion = sc.nextInt();
		sc.nextLine();
		// si la opción es cero cerrar el scanner y salir
		if (opcion == 0) {
			sc.close();
			return; // finaliza el método main
		}
		System.out.println();
		System.out.println("Introduce el primer operando");
		double a = sc.nextDouble();
		sc.nextLine();
		System.out.println("Introduce el segundo operando");
		double b = sc.nextDouble();
		sc.nextLine();
		// Variables para el resultado de la operación y la operación elegida
		double resultado = 0.0;
		String operacion = "";
		// Selección múltiple aplicada con switch
		switch (opcion) {
		case 1:
			operacion = " + ";
			resultado = a + b;
			break;
		case 2:
			operacion = " - ";
			resultado = a - b;
			break;
		case 3:
			operacion = " * ";
			resultado = a * b;
			break;
		case 4:
			operacion = " / ";
			resultado = a / b;
			break;

		default:
			operacion = "";
			break;
		}
		// Mostrar el resultado
		if (operacion.equals("")) {
			System.out.println("No se ha seleccionado una operación válida");
		} else {
			System.out.println(a + operacion + b + " = " + resultado);
		}
		// Cerrar Scanner de la consola
		sc.close();
	}
}
```

3.4 Sentencia goto

En Java, aunque la palabra está reservada no está implementada ninguna funcionalidad. En otros lenguajes se usa para interrumpir la ejecución del programa y seguir por donde se indique en dicha instrucción. De manera general, no se recomienda la utilización de esta instrucción como se discutirá más adelante en la unidad didáctica.

4 Nociones de diseño. Representación de algoritmos.

Vamos a empezar a estudiar qué pasos debemos seguir para elaborar un programa. Desarrollar un programa no es más que resolver un problema con unas determinadas herramientas. El primer paso será definir el problema que hay que resolver, el siguiente será el diseño del algoritmo que resuelve el problema y una vez diseñado se pasará a la elaboración del código que implemente dicho algoritmo.

El objetivo es introducir en este punto las herramientas que nos permitan resolver y representar algoritmos. El diseño del algoritmo será el paso previo al desarrollo de un programa.

La definición del problema es el primer paso del proceso a seguir para resolverlo. Hay que tener muy claro qué es lo que hay que resolver. Y esto implica una muy buena comunicación con la organización que encarga la realización del programa y con los usuarios finales, estableciendo con precisión y claridad los objetivos que se quieren alcanzar.

Normalmente, el punto de partida se establece cuando un cliente detecta que necesita resolver varias veces un problema de cierta complejidad y decide que será mucho más fácil si puede automatizar las operaciones mediante un programa. Llegados a esta conclusión, hay que empezar por la realización de un análisis del problema, que a veces puede llevar bastante tiempo, ya que el problema puede ser difícil de comprender o el cliente puede no tener del todo claro qué quiere.

Una vez el problema está definido, hay que llevar a cabo un **análisis funcional**. Esta no es estrictamente responsabilidad del programador, por lo que no se hará énfasis en este apartado, pero eso no implica que debamos conocer el proceso. A fin de cuentas, los programadores son los encargados de desarrollar los programas resultantes de esta etapa.

En este análisis, se acostumbra a hacer lo siguiente:

- Hablar con todos los usuarios implicados de manera que conseguimos tener todos los puntos de vista posibles sobre el problema.
- Dar todas las soluciones posibles con un estudio de viabilidad, considerando los costes económicos (material y personal).
- Proponer al cliente, futuro usuario de la aplicación, una solución entre las posibles, y ayudarle a tomar la decisión más adecuada.

- Dividir la solución adoptada en partes independientes para dividir las tareas entre un equipo de trabajo.

Como resultado, se dispondrá de una idea clara de **qué** hay que hacer y con qué recursos. Sólo entonces es el momento de ver **cómo** se debe hacer, en forma de una aplicación informática.

Una vez establecido cuál es el problema a resolver, es el momento de detenerse a reflexionar sobre qué estrategia seguir para resolverlo mediante un programa y qué forma deberá tomar el código fuente.

Un programa se desarrollará conforme a lo que se conoce formalmente como **algoritmo**: un mecanismo para resolver un problema o una tarea concreta, descrito como una secuencia finita de pasos.

Si bien todos los programas contienen algoritmos, traducidos en una serie de instrucciones, este concepto no está ligado exclusivamente a la programación. Cualquier descripción compuesta por una serie de pasos a seguir ordenadamente para alcanzar una meta es un algoritmo.

En cualquier caso, el hecho primordial es que, para establecer qué debe hacer un programa, antes hay que pensar exactamente como hay que dividir la tarea en acciones individuales y qué orden deben seguir. Por lo tanto, una de las tareas primordiales en programación antes de ni siquiera empezar a escribir código fuente es dedicar un tiempo a reflexionar y a diseñar un algoritmo que sirva para llevar a cabo la tarea dada por la definición del problema. Esta fase es muy importante, ya que condicionará totalmente la fase de implementación. Si el algoritmo es incorrecto se habrá perdido tiempo y esfuerzo creando un código fuente que no hace lo que se desea.

Acabamos de estudiar las **estructuras de control**, éstas permiten establecer qué instrucciones de su programa a ejecutar en cada momento y especificar el orden en que se ejecutan las instrucciones y qué caminos siguen en el proceso de ejecución. Es decir, establecer cuál es el **flujo de control** del programa. Estas estructuras nos permitirán resolver un problema con un programa y por lo tanto formarán parte del algoritmo que soluciona un problema.

Un algoritmo puede ser representado mediante muchos tipos de notaciones, veremos dos formas de hacerlo: mediante diagramas de flujo y mediante pseudocódigo.

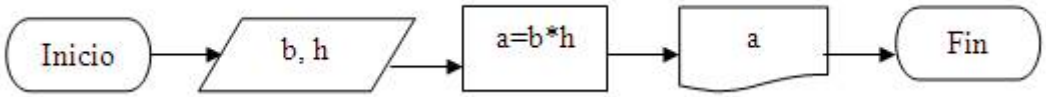
4.1 Diagramas de flujo

El **diagrama de flujo** es una representación gráfica de un algoritmo; representa gráficamente la secuencia lógica de las operaciones en la resolución de un problema. Debe reflejar el comienzo del programa, las operaciones, la secuencia en la que se realizan y el final del programa. Usan símbolos conectados con flechas para indicar la secuencia de instrucciones. Son usados para representar algoritmos pequeños, ya que abarcan mucho espacio y su construcción es laboriosa.

El conjunto de símbolos utilizados sigue unos estándares mundialmente aceptados y desarrollados por organizaciones tales como ANSI (American National Standards Institute) e ISO (International Standard Organization).

En el diagrama cada símbolo representa una acción en concreto; y cada instrucción del algoritmo se visualiza dentro del símbolo adecuado. Los símbolos se conectan con flechas para indicar el orden en que se ejecutan las instrucciones.

Por ejemplo, el siguiente diagrama de flujo corresponde al algoritmo para calcular el área del rectángulo:



Reglas para la construcción de diagramas de flujo:

1. Todo diagrama de flujo debe tener un inicio y un fin.

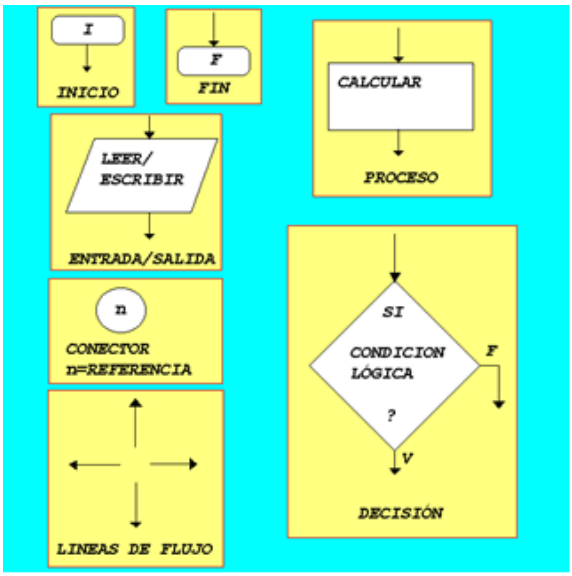
2. Las líneas de flujo nunca deben cruzarse, para evitarlo debe utilizarse el símbolo conector.

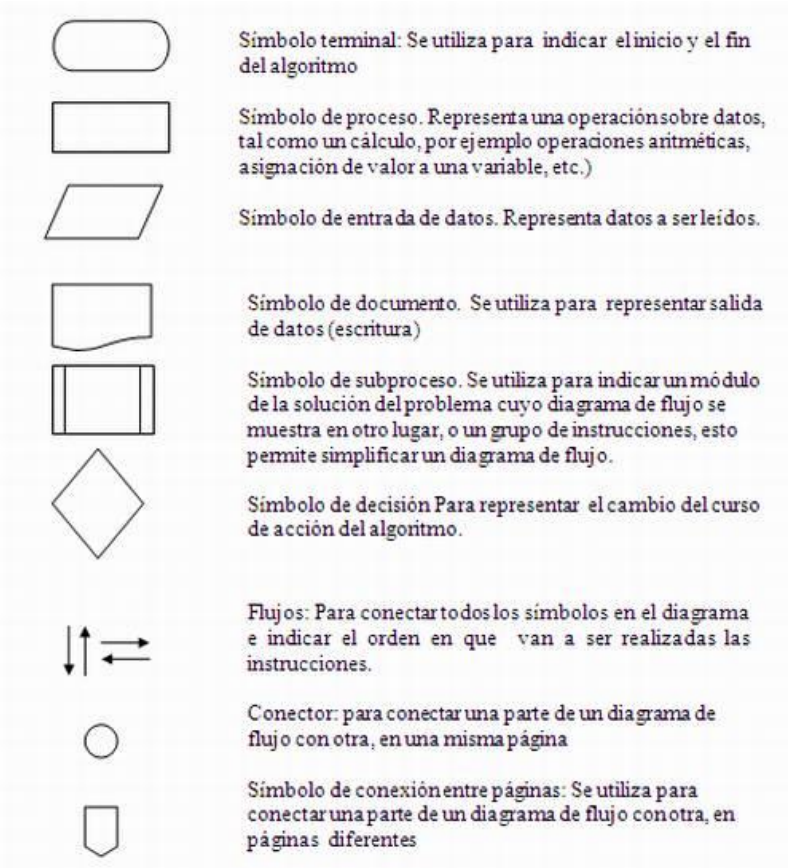
3. Las líneas de flujo deben terminar siempre en un símbolo.

4. Todos los símbolos en un diagrama deben estar conectados mediante una línea de flujo; todo símbolo debe tener una línea de flujo entrando y otra saliendo salvo el símbolo que indica inicio o fin del diagrama.

5. Como regla general el flujo del proceso debe mostrarse de izquierda a derecha y de arriba abajo.

Se recomienda mantener uniforme el tamaño de los símbolos, por lo que el texto que se escribe dentro no debe ser muy extenso, debemos recordar que el propio símbolo indica la operación a realizar. La forma en que se capturan los datos de entrada o se muestran los datos de salida se detallarán al codificar el algoritmo en el lenguaje de programación.





4.2 Pseudocódigo

El **pseudocódigo** es una descripción de alto nivel compacta e informal del principio operativo de un algoritmo. Utiliza las convenciones estructurales de un lenguaje de programación real, pero está diseñado para la lectura humana en lugar de la lectura mediante máquina, y con independencia de cualquier lenguaje de programación. Normalmente, el pseudocódigo omite detalles que no son esenciales para la comprensión humana del algoritmo, tales como declaraciones de variables, código específico del sistema y algunas subrutinas. El lenguaje de programación se

complementará, cuando sea conveniente, con descripciones detalladas en lenguaje natural, o con notación matemática compacta. Se utiliza pseudocódigo ya que este es más fácil de entender para las personas que el código del lenguaje de programación convencional. Se utiliza comúnmente en los libros de texto y publicaciones científicas que se documentan varios algoritmos, y también en la planificación del desarrollo de software para esbozar la estructura del programa antes de realizar la efectiva codificación.

No existe una sintaxis estándar para el pseudocódigo.

Ejemplos de representación manejo de datos:

Salida: *Escribir VARIABLE*

Entrada: *Leer VARIABLE*

Asignación: *VARIABLE1 :=VARIABLE2*

o

VARIABLE1 <- VARIABLE2

Para representar estructuras de control cada autor utiliza sus propias convenciones del lenguaje, por lo que existen muchas y diversas. Las principales son:

- Secuencial:

  Instrucción 1

  Instrucción 2

  …

- Selectiva:

  Si P Entonces

  …

  Sino

  …

  Fin Si

- Selección múltiple:

  Seleccionar P,

  Caso Valor1:

  …

  Caso Valor2:

  …

  En otro caso:

  …

  Fin Seleccionar

- Iterativa:

  Mientras P

  Hacer …

  Fin Mientras

Para el ejemplo anterior de cálculo del área de un rectángulo:

Inicio

Leer Base, Altura

*Área = Base * Altura*

Escribir Área

Fin

5 Programación Estructurada

Hasta ahora habíamos creado programas que ejecutan las instrucciones de código una a una, de principio a fin. En esta unidad hemos estudiado las estructuras de control disponibles en los lenguajes de programación que nos permiten ejecutar de manera condicional una serie de acciones y las estructuras repetitivas que permiten entrar en un bucle y repetir una acción un número determinado de veces o hasta que se cumpla una determinada condición. Además de las estructuras de salto que permiten que el flujo de programa no sea lineal si no que puede ir de un punto de código a otro, en muchos lenguajes de programación existe la sentencia ***GoTo*** que permite transferir la ejecución del programa a otra línea de código distinta de la siguiente en orden en el código fuente.

Estas estructuras dotan a los programas de una flexibilidad que puede provocar que los programas puedan construirse de manera que sea muy difícil predecir su comportamiento y seguir su flujo (*spaghetti code*) si no se siguen determinadas reglas. Para evitar estas situaciones surgió la programación estructurada.

La **programación estructurada** es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa recurriendo únicamente a subrutinas y tres estructuras básicas:

- **Secuencia**; las instrucciones se ejecutan en el mismo orden en que se han escrito.
- **Selección** (*if y switch*); hay ciertas condiciones que provocan la ejecución de bloques de instrucciones diferentes dependiendo de si se cumplen o no estas condiciones.
- **Iteración** (bucles *for y while*); el bloque de instrucciones se ejecuta un número finito de veces, ya sea un número concreto o hasta que se cumple una condición.

Asimismo, se considera innecesario y contraproducente el uso de la instrucción de transferencia incondicional (*GOTO*), que podría conducir a código espagueti, mucho más difícil de seguir y de mantener, y fuente de numerosos errores de programación.

El teorema del programa estructurada, propuesto por Böhm-Jacopini, demuestra que todo programa puede escribirse utilizando únicamente las tres instrucciones de control siguientes:

- Secuencia
- Instrucción condicional
- Iteración (bucle de instrucciones) con condición inicial

El teorema del programa señala que la combinación de las tres estructuras básicas, secuencia, selección e iteración, son suficientes para expresar cualquier función computable.

Ventajas de la programación estructurada

Entre las ventajas de la programación estructurada sobre el modelo anterior (hoy llamado despectivamente código espagueti), cabe citar las siguientes:

- Los programas son más fáciles de entender, pueden ser leídos de forma secuencial y no hay necesidad de tener que rastrear saltos de líneas (*GOTO*) dentro de los bloques de código para intentar entender la lógica interna.
- La estructura de los programas es clara, puesto que las instrucciones están más ligadas o relacionadas entre sí.
- Se optimiza el esfuerzo en las fases de pruebas y depuración. El seguimiento de los fallos o errores del programa (depuración), y con él su detección y corrección, se facilita enormemente.
- Se reducen los costes de mantenimiento. Análogamente a la depuración, durante la fase de mantenimiento, modificar o extender los programas resulta más fácil.
- Los programas son más sencillos y más rápidos de confeccionar.
- Se incrementa el rendimiento de los programadores.

La recomendación general es, pues, seguir los postulados de la programación estructurada y hacer nuestros programas los más claros y fáciles de seguir que

podamos. Este debate sobre programación estructurada se produjo en los años 60 y principios de los 70, con lo que históricamente está completamente aceptado y superado. De hecho, Java que es un lenguaje publicado ya en 1995 ni siquiera implementa la sentencia *goto* aunque sí la mantiene como palabra reservada sin ninguna funcionalidad.

6 Control de excepciones

Hemos visto hasta ahora las estructuras de control de flujo del programa *if, while, for, return, break*, etc., estas estructuras nos permiten definir el flujo de la ejecución del programa en función de ciertas condiciones. En cambio, ninguna de ellas tiene en cuenta que se puedan producir errores en tiempo de ejecución de un programa y como sabemos se producen errores durante las ejecuciones de los programas.

Veamos ahora la funcionalidad básica que ofrece Java para el tratamiento de estos errores. Posteriormente, a lo largo del curso, trataremos más en profundidad el tratamiento de errores.

Cuando en Java se produce un error en Java se "lanza una excepción", estas excepciones son objetos de tipo ***Exception***, estos objetos, excepciones, contienen la información del error que se ha producido: excepción aritmética, fichero no encontrado, división por cero, etc.

La estructura disponible en java para capturar estas excepciones es el bloque ***try / catch / finally***, tiene la siguiente sintaxis:

```
try {
    // sentencias que pueden causar una excecpción
}
catch (Exception e) {
   // manejo de la excepción
}
finally {
   // sentencias de finalización
}
```

Bloque *try*

Try en inglés es el verbo intentar, así que todo el código que vaya dentro de esta sentencia será el código sobre el que se intentará capturar el error si se produce y una vez capturado hacer algo con él. Lo ideal es que no ocurra un error, pero en caso de que ocurra un bloque try nos permite estar preparados para capturarlo y tratarlo.

Bloque *catch*

En este bloque definimos el conjunto de instrucciones necesarias o de tratamiento del problema capturado con el bloque *try* anterior. Es decir, cuando se produce un error o excepción en el código que se encuentra dentro de un bloque *try* la ejecución del programa irá al bloque *catch* donde tendremos disponible la información de la

excepción que se ha producido; fijémonos que después de *catch* hemos puesto unos paréntesis donde pone "*Exception e*". Esto significa que cuando se produce un error Java genera un objeto de tipo ***Exception*** con la información sobre el error y este objeto se envía al bloque *catch*.

Es posible tener varios bloques catch para el mismo bloque *try*, con distintos tipos de excepciones: *ArithmeticException, IOException* etc. De manera que podemos reaccionar de maneras diferentes dependiendo del tipo de excepción producida. El tipo *Exception* representa cualquier tipo de excepción que se pueda producir en Java. Cuando se produzca una excepción la ejecución del programa entrará en el primer bloque cuyo tipo de excepción se ajuste al tipo de excepción producido e ignorará el resto. Por lo tanto, si vamos a usar varios bloques *catch* y uno va a tener el tipo *Exception*, éste deberá ser el último.

Bloque *finally*

Por último, tenemos el bloque *finally* que es un bloque donde podremos definir un conjunto de instrucciones necesarias tanto si se produce error o excepción como si no y que por tanto se ejecuta siempre. Típicamente, aquí pondremos código que permitirá liberar memoria, cerrar ficheros, conexiones, etc.

El bloque *finally* es opcional y si no necesitamos hacer ninguna acción de cierre para responder a la excepción podemos omitirlo.

En el siguiente ejemplo se pueden producir fácilmente 2 excepciones: la primera que el dato introducido por consola no corresponda con el tipo que estamos intentando recoger (ya hemos visto maneras de asegurar el formato) y la segunda es una división por cero. Prueba la ejecución del programa en ambas situaciones y comprueba que se ejecuta el bloque *finally*.

```java
import java.util.Scanner;
public class Calculadora {
	public static void main(String[] args) {
		// variable scanner
		Scanner sc = null;
		//bloque try, este código podría generar errores
		try {
			// Inicializar objeto Scanner para leer de la consola
			sc = new Scanner(System.in);
			// Mostrar información al usuario y recoger valores desde el teclado
			System.out.println("Calculadora");
			System.out.println("1 - Suma");
			System.out.println("2 - Resta");
			System.out.println("3 - Producto");
			System.out.println("4 - Cociente");
			System.out.println("5 - Módulo");
			System.out.println("Elige una opción (0 para salir)");
			int opcion = sc.nextInt();
			sc.nextLine();
			if (opcion == 0) {
				sc.close();
				return;
			}
			System.out.println();
			System.out.println("Introduce el primer operando");
			int a = sc.nextInt();
			sc.nextLine();
			System.out.println("Introduce el segundo operando");
			int b = sc.nextInt();
			sc.nextLine();
			// Variables para el resultado de la operación y la operación elegida
			int resultado = 0;
			String operacion = "";
			// Selección múltiple aplicada con switch
			switch (opcion) {
			case 1:
				operacion = " + ";
				resultado = a + b;
				break;
			case 2:
				operacion = " - ";
				resultado = a - b;
				break;
			case 3:
				operacion = " * ";
				resultado = a * b;
				break;
			case 4:
				operacion = " / ";
				resultado = a / b;
				break;
			case 5:
				operacion = " % ";
				resultado = a % b;
				break;
			default:
				operacion = "";
				break;
			}
			// Mostrar el resultado
			if (operacion.equals("")) {
				System.out.println("No se ha seleccionado una operación válida");
			} else {
				System.out.println(a + operacion + b + " = " + resultado);
			}
		} catch (Exception e) {

			System.out.println();
			System.out.println("Se ha producido un error inesperado");
			e.printStackTrace(); //muestra la información de la excepción
		} finally {
			// Cerrar Scanner de la consola si sc es distinto de null
			if(sc !=null) {
				sc.close();
			}
			System.out.println("Final de ejecución de la Calculadora");
		}
	}
}
```