



## **Ingeniería en Desarrollo de Software**

**3<sup>er</sup> semestre**

Programa de la asignatura:

**Programación orientada a objetos I**

**Unidad 2**

**Métodos y estructuras de control**

**Clave:**

**Ingeniería: TSU:**

**15142316 / 16142316**

**Universidad Abierta y a Distancia de México**





### Índice

Unidad 2. Métodos y estructuras de control .....	3
Presentación de la unidad .....	3
Propósito .....	3
Competencia específica.....	3
Consideraciones específicas de la unidad: actividades, requerimientos de asignatura, otras .....	4
2.1. Generalidades de los métodos.....	4
2.1.1. Declaración y ámbito de variables.....	5
2.1.2. Declaración de constantes .....	10
2.2. Métodos que no devuelven valores.....	12
2.2.1. Características .....	12
2.2.2. Llamada de métodos.....	12
2.3. Métodos que devuelven valores.....	13
2.3.1. Recepción de parámetros .....	14
2.3.2. Retorno de parámetros .....	15
2.3.3. Paso de parámetros.....	15
2.3.4. Llamada de métodos.....	16
2.4. Estructuras de control selectivas.....	17
2.4.1. If-else.....	17
2.4.2. If anidado .....	19
2.4.3. Switch-case .....	20
2.5. Estructuras de control cíclicas.....	23
2.5.1. While.....	23
2.5.2. Do-while.....	24
2.5.3. For .....	25
Cierre de la unidad .....	29
Para saber más .....	30
Fuentes de consulta .....	30



## Unidad 2. Métodos y estructuras de control

### Presentación de la unidad

En la unidad anterior se vieron temas introductorios para la programación OO, cuyos temas fungieron como iniciación para comprender el uso y aplicación de la programación. En esta unidad de POO I, aprenderás el concepto de los métodos y su utilización para crear programas modulares, así como el manejo de las estructuras de control selectivas y cíclicas. Todo esto se utilizará para que realices programas más completos y complejos en la siguiente unidad, sobre la aplicación de las características de POO y las excepciones.

### Propósito



Al término de esta unidad lograrás:

- Describir las generalidades de los métodos.
- Distinguir los métodos que devuelven valores contra los que no devuelven.
- Manejar las estructuras de control selectivas y cíclicas.

### Competencias específicas



- Desarrollar programas modulares para solucionar problemas diversos, mediante la aplicación de los diferentes tipos de variables, métodos, y estructuras de control en lenguaje Java.

### Consideraciones específicas de la unidad: actividades, requerimientos de asignatura, otras

Es muy importante que captures todos los programas de ejemplo, para que analices su sintaxis y puedas comprender mejor los temas vistos.

#### 2.1. Generalidades de los métodos

Cuando se está trabajando en el lenguaje de programación Java, la clase es el elemento básico que lo conforma, pero a su vez está conformada por varios subbloques de código que la conforman, los cuales se llaman métodos y agrupan de manera más o menos ordenada a los algoritmos que conforman el cuerpo de nuestro programa o aplicación.



Todos los métodos tienen cierta estructura que los distinguen de los demás elementos del lenguaje Java, la estructura clásica de un método es la siguiente:

```
tipo_devuelto nombre_método([parámetros]) {  
    sentencia1;  
    sentencia2;  
    sentenciaN;  
}
```



Nótese que el elemento `[parámetros]` está delimitado por corchetes, lo que quiere decir que es un elemento opcional. Cuando los parámetros se manejan como lista de ellos, o sea más de uno, se separa esta lista con una coma entre cada uno.

Además de lo anterior, podremos entender a los métodos como:

- Un bloque de código que tiene un nombre.
- Capaces de recibir parámetros o argumentos (opcionalmente, como ya se dijo).
- Contienen sentencias que le permiten realizar la tarea para la cual fue creado.
- Devuelven un valor de algún tipo (recordar el tema de tipo de datos) conocido.

### 2.1.1. Declaración y ámbito de variables

Haciendo referencia al apartado de tipos de datos, se hablará ahora del concepto de variable que se entenderá como una locación de memoria reservada para alojar un dato que cambiará con respecto al tiempo/espacio donde se ejecute la aplicación. Tienen asociado un identificador y un tipo de dato que le da nombre (identificador) e indica qué guardaremos en ella (tipo de dato); lo que es aplicable a Java y cualquier otro lenguaje de programación.

En Java, cuando se va a hacer uso de una variable primero se debe declarar, es decir, indicarle al compilador que hemos reservado esa locación de memoria para poder alojar un dato de interés para nuestra aplicación. De no ser así, y aunque pongamos un identificador a la variable, Java nos indicará que no está definida.

En función a los datos que almacenan las variables se clasifican en:

- Variables primitivas: sirven para almacenar datos numéricos, valores lógicos o caracteres.
- Variables referenciadas: asociadas a un objeto, o son la instancia de una clase.

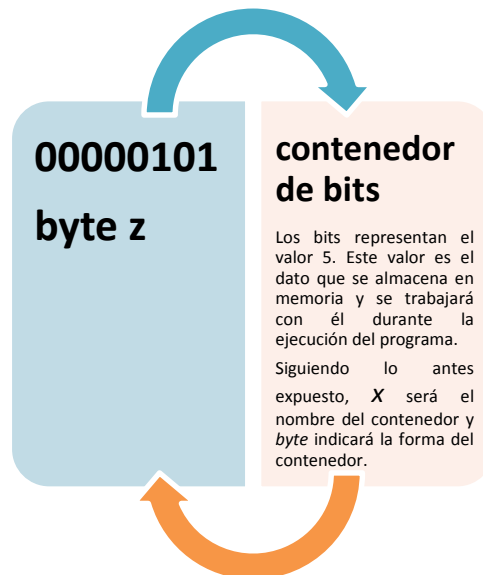


Para profundizar un poco más en el concepto, se dirá que una variable no es nada más que un contenedor de bits que representan un valor. Se entenderá de la siguiente manera:

- Cuando se habla de variables primitivas, los bits que la forman sólo representan un número que coincide con el valor asignado a la misma y. como se indicó, puede quedarse conteniendo ese valor o puede cambiar a lo largo de la ejecución del programa. Por ejemplo, relacionando lo expuesto ahora con el tema de tipos de dato, se tiene que hay variables de tipo byte (con una longitud de 8 bits en memoria) y puede almacenar números enteros con signo entre -128 y 127 (cuando no se utiliza el signo el valor puede ser hasta 255), de tipo int (con una longitud de 32 bits en memoria), para almacenar números enteros con signo entre -2,147,483,648 al 2,147,483,647, de tipo float con sus respectivos límites hasta abarcar todos los tipos de dato.

Visto de manera gráfica pudiera quedar como sigue:

`byte z = 5; // declaración de variable de tipo byte válida en Java`



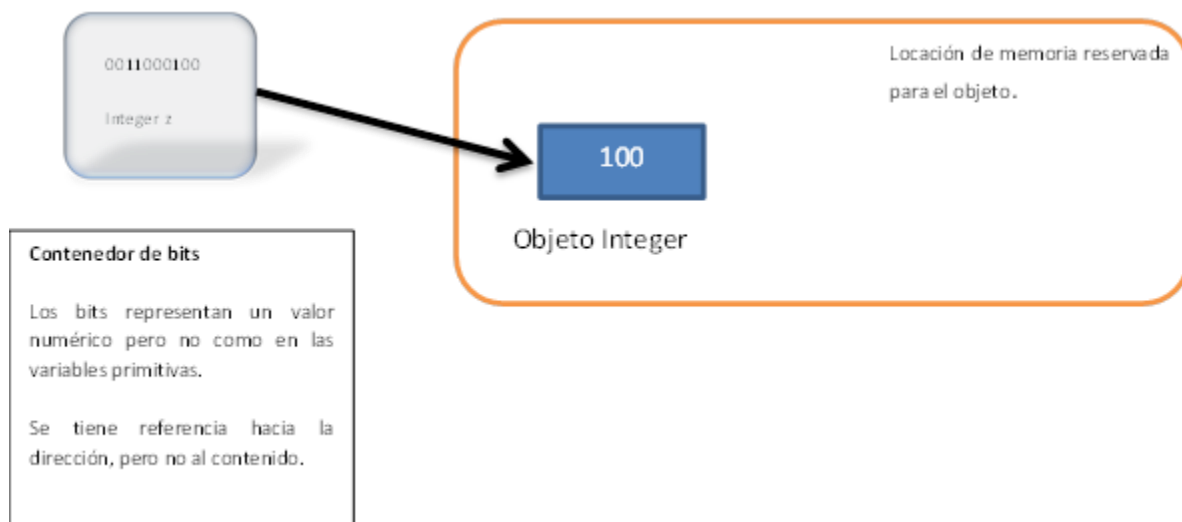




Cuando se habla de variables referenciadas o asociadas a un objeto, los bits representan un identificador que permite acceder al valor de la variable; es decir, nos da una referencia al objeto, pero no es el objeto a la variable en sí.

Visto de manera gráfica, pudiera quedar como sigue:

`Integer z=new Integer(100);` //declaración de una variable de tipo objeto Integer, válido en Java.



Se muestra a continuación algunos ejemplos de declaraciones de variables referentes a Java, para los tipos de variables primitivas, referenciadas o asociadas a un objeto:

```
int a; // declaración de una variable 'a' de tipo entero
//su valor inicial (por definición) es 0.
```

```
int b = 10;
// declaración de una variable 'b' de tipo entero
// con valor inicial establecido a 10.
```

```
Nombre_Clase referencia;
// declaración de una variable con identificador 'referencia'
```



```
// donde su tipo de dato será 'Nombre_Clase' o dicho de otra
manera
// aquí es donde se crea el contenedor.

Nombre_Clase referencia2;
// mismo caso que la variable anterior,
// pero el identificador ahora es 'referencia2'.
referencia = new Nombre_Clase;
// En este tipo de variables la declaración no es suficiente se
deberé crear
// un nuevo objeto de la clase 'Nombre_Clase', y es asignado a la
variable
// 'referencia'. En el paso anterior se dijo que se crea el
contenedor,
// en este paso se le asigna un valor (referencia al objeto) al
contenedor.

referencia2 = referencia;
// Ahora también 'referencia2' tiene el mismo
// objeto a su cargo que 'referencia'
```

Ahora que se ha aclarado un poco la idea de qué es una variable, sus componentes principales y el modo en que se hace una declaración de la misma, debemos considerar las distintas partes del programa donde la utilizaremos. Es lo que se conoce como ámbito de la variable.

Este concepto es uno de los más importantes que debemos conocer cuando estamos trabajando y manipulando variables. Se llama ámbito de una variable al lugar donde ésta está disponible. Por lo general, cuando declaramos una variable, sólo estará disponible en el bloque de código (encerrado entre llaves) donde se ha declarado, que puede ser desde el ámbito de la clase completa, el ámbito sólo de un método, o incluso dentro del ámbito de los elementos que lo conforman, por ejemplo las sentencias *if*, *while*, *for* y entre todos los tipos disponibles que tenga Java.





Por ejemplo:

```
public class AmbitoVariables{
    public static void main(String[] args){
        if (true) {
            int a=10; //Decalaración de la variable dentro del bloque que
corresponde
            // a la sentencia if (dentro del ámbito del if).
        }
        System.out.println(a); // Acceso a la variable fuera del ámbito
donde
        //fue declarada (fuera del bloque if, delimitado
        //por llaves).
    }
}
```

Al compilar esta pequeña clase de ejemplo, se notará que el compilador de Java lanza un mensaje de error que indicará que la variable a la que se hace referencia (variable con identificador a) no ha sido encontrada; en otras palabras, su declaración no está dentro del ámbito en donde se pretende usar. El mensaje de error del compilador de Java tendrá un aspecto como se muestra en las siguientes imágenes.

```
12 public class Main {
13     /**
14      * @param args the command line arguments
15      */
16     public static void main(String[] args) {
17         // TODO code application logic here
18         if(true){
19             int a = 10;
20             System.out.println(a);
21         }
22     }
23 }
24
25
```

Figura 1

```
12 public class Main {
13     /**
14      * @param args the command line arguments
15      */
16     public static void main(String[] args) {
17         // TODO code application logic here
18         if(true){
19             int a = 10;
20             System.out.println(a);
21         }
22     }
23 }
24
25
```

Figura 2



Se puede notar claramente que, en la primer imagen (Figura 1), el IDE nos está indicando algún tipo de error en la línea 21, señalando la variable con identificador *a* como la posible causa. En la segunda imagen (Figura 2) nos da los detalles completos de la causa del error, el programador deberá entender que no se puede encontrar el símbolo (variable) especificado.

Hay más normas de ámbito respecto a las variables miembro de una clase, donde intervienen los modificadores de acceso *public* o *private*. Cuando una variable es declarada con el modificador *public*, se accede directamente a ellas a través de `nombre_clase.nombre_variable`. En el caso de que una variable sea declarada con el modificador *private*, sólo se podrá acceder a ella mediante los métodos de la clase que lo permitan.

### 2.1.2. Declaración de constantes

Desde la concepción de la definición del lenguaje Java, no fueron consideradas las constantes en su término más estricto, entendidas como un valor que no puede ser alterado durante la ejecución de un programa, y corresponde a una longitud fija en la memoria de la computadora donde se ejecuta el programa.

Sin embargo, para hacer frente a esta pequeña inconveniencia del lenguaje, una declaración de variable como *static final*, se puede considerar una constante muy efectiva. El modificador *static* permite el acceso a una variable miembro de una determinada clase, sin haber cargado una instancia de ésta donde será utilizada. El modificador *final* provoca que la variable (su contenido) no pueda ser cambiada.

La siguiente declaración define la constante llamada *PI* cuyo valor está definido por el número  $\pi$  (3.1415927...).

```
static final float PI = 3.1415927; //Constante que representa el número PI.
```



Como ya se dijo, si se intenta modificar el valor contenido en la variable identificada como PI, se dará un error de compilación.



### 2.2. Métodos que no devuelven valores

Una vez que se estudiaron las generalidades sobre los métodos, se deberá conocer que existen dos tipos cuando se habla de ellos:

- Métodos que retornan valores.
- Métodos que no retornan valores.

En este apartado se analizarán los métodos que no regresan valores.

#### 2.2.1. Características

El hecho de que un método regrese valores o no es opcional. Cuando un método no retorna valores, se deberá modificar la definición de sus parámetros y quedará como sigue:

```
void nombre_método([parámetros]){  
    sentencia1;  
    sentencia2;  
    sentenciaN;  
}
```

En donde se deberá utilizar la palabra reservada *void* que le indica al compilador y al método que su valor de retorno será “vacío”. Dentro del cuerpo de sus sentencias se omite *return*, por no “regresar” valor alguno a donde fue llamado. Los parámetros de igual manera son opcionales, aunque los paréntesis van por obligación.

#### 2.2.2. Llamada de métodos

Cuando decimos que llamamos un método, estamos haciendo uso de las sentencias que lo conforman. De igual manera, se dice que se invoca al método.



Los métodos se invocan (llaman) por el nombre definido en su declaración, y se deberá pasar la lista de argumentos entre paréntesis. Por ejemplo, se definirá un método que imprima a pantalla un mensaje para el usuario.

```
public void saludo_usuario() {  
    String cadena_Saludo = "Hola usuario, feliz viaje en Java";  
    System.out.println(cadena_Saludo);  
    System.out.println("Mensaje impreso desde un método");  
}
```

Cuando ya se ha definido el método y sus sentencias son las necesarias y adecuadas, se hace una invocación a él o, mejor dicho, se invocan las sentencias que conforman el cuerpo del método. Se realiza de la siguiente manera:

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

La salida será la siguiente:

```
Hola usuario, feliz viaje en Java  
Mensaje impreso desde un método
```

### 2.3. Métodos que devuelven valores

Por definición del lenguaje Java, todos los métodos deberían regresar valores, excepto en los casos explicados en el apartado anterior. Cuando los métodos declarados en Java regresan valores, quiere decir que se espera recibir alguna respuesta o resultado de alguna operación realizada por las sentencias que conforman su cuerpo. Para que un método pueda regresar valores, deben estar involucrados varios factores que le ayuden o “alimenten”, para poder realizar dichas operaciones. Estos elementos básicos son:



- Parámetros de entrada, que son las entradas propias del método que le ayudarán a saber sobre qué se realizarán las operaciones indicadas en las sentencias que lo conforman, aunque este apartado se puede considerar opcional.
- Parámetros de retorno, que es la respuesta que nos regresará una vez que haya ejecutado el cuerpo de sus sentencias.

En los apartados siguientes se explicará, con más detalle, los elementos mencionados y algunos otros que son necesarios para los métodos que regresan valores.

### 2.3.1. Recepción de parámetros

Los parámetros de un método se pueden entender como los valores que éste recibe desde la parte del código donde es invocado. Los parámetros pueden ser de tipos primitivos (*int*, *double*, *float*, entre otros) u objetos.

Como ya se ha visto, en la declaración del método se escribe qué parámetros recibirá dentro de los paréntesis, separando cada uno de ellos por una coma, mencionando el tipo de cada uno de ellos y el identificador, se entenderá también que no hay límites en el número de parámetros que se envían al método. Se presenta un ejemplo:

```
public int obtener_potencia(int base, int potencia){  
    ...  
}
```

Este método recibe dos parámetros de tipo entero, cuyos identificadores son *base* y *potencia*.





### 2.3.2. Retorno de parámetros

Cuando el método de nuestro interés ya ha recibido (o consigue) los elementos necesarios para poder realizar sus funciones, retornará el resultado de aplicarlas sobre los parámetros recibidos. Se deberá indicar en la declaración del método qué tipo de dato retornará, y señalar en el cuerpo de sus sentencias la palabra reservada *return* (con esto se precisa que regresará). Se ilustra un ejemplo a continuación:

```
public int obtener_potencia(int base, int potencia){  
    int resultado = Math.pow(base, potencia);  
    return resultado;  
}
```

Las partes del código que están resaltadas indican el retorno de parámetros del resultado de obtener la potencia de los parámetros recibidos (base y potencia). El lector deberá notar que el tipo de dato declarado en la definición del método es el mismo que el utilizado en la variable de retorno (resultado), si fuera de otra manera, el compilador lanzará un error de tipos incompatibles.

### 2.3.3. Paso de parámetros

Cuando se invoque (llame) el parámetro que se quiera utilizar, deberemos pasar los indicados (necesarios) en su declaración. Los parámetros se escriben a continuación del nombre del método que se quisiera utilizar, de igual manera entre paréntesis, pero al hacer la invocación no se indica el tipo de dato de los mismos, aunque las variables que enviemos forzosamente deberán coincidir con el tipo que pretendemos utilizar.

Por ejemplo, se completará el llamado del método presentado en el punto anterior.

```
public static void main(String[] args){  
    int b = 3; //representando a la base
```



```
int p = 2; //representando a la potencia
obtener_potencia(b,p);
...
}
```

Como información al lector, se debe aclarar que se omiten muchas partes de los métodos, sólo son ejemplos ilustrativos de cada uno de los puntos.

También como punto importantísimo, recordando los tipos de variables, se dirá que en Java los parámetros de tipo primitivo (`int`, `long`, `double`, ...) siempre se pasan por valor. Las variables de tipo objeto y *arrays* se pasan por referencia (a su dirección de memoria de alojamiento).

### 2.3.4. Llamada de métodos

Para complementar todos los elementos de los métodos que regresan valores, se indicará cuál es la forma correcta de llamar a los métodos, pasando los parámetros correctos y recibiendo de manera adecuada los valores de retorno.

Como ya se mencionó, al invocar un método simplemente se deberá escribir el nombre del que interesa utilizar, seguido de los parámetros que necesita (fueron indicados en su declaración). Como ejemplo, se terminará de desarrollar el código del método de la obtención de potencias.

```
public static void main(String[] args){
    int b = 3; //representando a la base
    int p = 2; //representando a la potencia
    int r_potencia // variable que recibirá el resultado desde el
    método
    r_potencia = obtener_potencia(b,p);
    System.out.println("El resultado es: "+r_potencia);
}
```



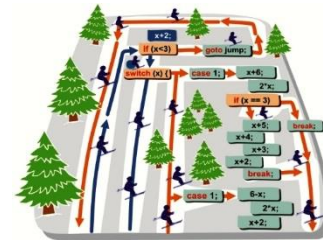
Donde:

- `r_potencia` será el valor de recepción del parámetro devuelto por el método. De igual manera deberá coincidir el tipo de dato de la variable de recepción, con el tipo de dato indicado en el mismo. En este caso indica que regresará un `int`, entonces el tipo de la variable deberá ser también `int`.
- La llamada al método `obtener_potencia` será la conjunción de todos los puntos explicados, necesitará un lugar donde depositar su valor de retorno (explicado en el punto anterior), y pasar los parámetros necesarios después del nombre del método, sin olvidar que todos los tipos de las variables deben coincidir (las variables que se pasan como parámetros, la que se retorna desde el método, y aquella donde se recibe este valor de retorno). Se representa en esta línea:

```
r_potencia = obtener_potencia(b,p);
```

## 2.4. Estructuras de control selectivas

Las estructuras alternativas son construcciones que permiten alterar el flujo secuencial de un programa, de forma que en función de una condición o el valor de una expresión, el mismo pueda ser desviado en una u otra alternativa de código (García, Rodríguez, Mingo, Imaz, Brazález, Larzabal et ál., 2000).



### 2.4.1. If - else

La estructura de selección simple o *if* es aquella secuencia de un programa que se ve afectada por una condición, si ésta es cumplida la secuencia del programa entrará en el bloque de instrucciones de dicha estructura, de lo contrario la secuencia del programa se “saltará” ese bloque de código y entrará al bloque de instrucciones *else*, dado que la condición se cumplió. Esta estructura de control se conforma como sigue:

```
// Instrucciones previas a la condición
if( condición) {
    Instrucciones a ejecutarse solo si la condición se cumplió.
}
```



```
else{  
    Instrucciones a ejecutarse si la condición NO se cumplió.  
}  
// Instrucciones a ejecutarse después de la condición
```

Las llaves se utilizan para agrupar las instrucciones que se han de ejecutar, si sólo quieres que se ejecute una instrucción puedes omitirlas. Es importante mencionar que las instrucciones previas y posteriores al bloque *if-else* se ejecutan de manera normal, se cumpla o no la condición, por lo que sólo se altera la secuencia del programa en ese bloque.

A continuación se muestra un ejemplo donde dada una calificación, se decide si es aprobatoria o no, y esto se indica en un mensaje.

<pre>package ejemplos; /**  * @author ESAD  */ public class estructuralF {      public static void main(String[] args) {         int calificacion=80;          if(calificacion&gt; 70){             System.out.print("Aprobado");         }         else{             System.out.print("Reprobado");         }         System.out.print("esto se imprimirá se         cumpla o no la condición");     } }</pre>	<p>Define el paquete donde se encuentra</p> <p>Indica el nombre del autor</p> <p>Declara la clase llamada estructuralF</p> <p>Se declara el método main</p> <p>Declaro la variable calificación</p> <p>Establezco la condición de <i>If</i></p> <p>Se muestra a pantalla el resultado</p> <p>Palabra reservada <i>else</i></p> <p>Se envía el mensaje a pantalla.</p> <p>Se manda una impresión que se ejecutará después de la estructura selectiva <i>if –else</i>.</p>
---	--



### 2.4.2. If anidado

Esta estructura es un agrandamiento de la anterior, ya que se pueden evaluar diferentes condiciones ligadas.

Por ejemplo, en el tema anterior sólo queríamos que se mostrara Aprobado o Reprobado, según la calificación, pero ahora se quiere dar mayor reconocimiento a los puntos obtenidos. Esto se realizaría de la siguiente manera:

```
package ejemplos;
/**
 * @author ESAD
 */
public class estructuraIF {

    public static void main(String[] args) {
        int calificacion=90;

        if(calificacion> 90 && calificacion <= 100){
            System.out.print("Excelente");
        }
        else if(calificacion >80 && calificacion <= 90){
            System.out.print("Muy Bien");
        }
        else if(calificacion >70 && calificacion <= 80){
            System.out.print("Bien");
        }
        else if(calificacion < 70){
            System.out.print("Muy mal");
        }
    }
}
```



En este ejemplo evaluamos que, si su calificación está entre 90 y 100, se mostrará un mensaje diciendo “Muy bien”. Si no tienes una calificación entre 90 y 100, pero está entre 80 y 90, se mostrará “Bien”; pero si no cumplió ninguna de las anteriores, y la calificación es menor de 70 se mostrará “Muy mal”.

Como has podido observar, se pueden colocar condiciones múltiples como en este ejemplo, donde no sólo se evaluó una calificación, sino todo un rango, para ello se auxilió de los operadores lógicos y relacionales que se vieron en la unidad anterior.

### 2.4.3. Switch-case

La estructura selectiva *switch-case* también es conocida como de decisión múltiple. En ésta se puede evaluar una misma expresión con, como su nombre lo indica, múltiples opciones, pero en este caso no son condiciones, el switch evalúa si una variable tiene un valor en específico. A continuación se muestra una estructura:

```
// Instrucciones previas a la condición
switch(expresion) {
    case valor: instrucciones a ejecutarse;
    break;
    case valor: instrucciones a ejecutarse;
    break;
    default: instrucciones;
}
// Instrucciones a ejecutarse después de la condición
```

Una sentencia *case* es una opción de los diferentes valores que puede tomar la expresión que se evalúa. Por cada valor permitido que la expresión pueda tomar, se representará una sentencia *case*.

Las características más relevantes de *switch* son las siguientes (Gosling, Holmes y Arnold, 2001):





1. Cada sentencia *case* se corresponde con un único valor. No se pueden establecer rangos o condiciones sino que se deben comparar con valores concretos.
2. Los valores no comprendidos en ninguna sentencia *case* se pueden gestionar en *default*, que es opcional.
3. En la ausencia de *break*, cuando se ejecuta una sentencia *case* se ejecutan también todas las *case* que van a continuación, hasta que se llega a un *break* o termina el *switch*.

Se muestra un ejemplo a continuación:

```
package ejemplos;

import javax.swing.JOptionPane;

/**
 * @author ESAD
 */
public class ejemploSwitch {

    public static void main(String[] args) {

        int mes= Integer.parseInt(JOptionPane.showInputDialog("Cuál es el
número del mes"));

        switch(mes){
        case 1: System.out.print("Enero");
        break;
        case 2: System.out.print("Febrero");
        break;
        case 3: System.out.print("Marzo");
        break;
        case 4: System.out.print("Abril");
        break;
```



```
case 5: System.out.print("Mayo");
break;
case 6: System.out.print("Junio");
break;
case 7: System.out.print("Julio");
break;
case 8: System.out.print("Agosto");
break;
case 9: System.out.print("Septiembre");
break;
case 10: System.out.print("Octubre");
break;
case 11: System.out.print("Noviembre");
break;
case 12: System.out.print("Diciembre");
break;
default: System.out.print("Mes no válido");
}
}
```

Este ejemplo muestra que, al inicio del programa, se crea una variable llamada *mes*, la cual almacena el número del mes indicado por el usuario, y muestra en la pantalla el nombre del mes correspondiente. Para identificarlo utilizamos la sentencia `switch`, que para este ejemplo tiene 12 casos que son los meses que existen. Al encontrar el valor correspondiente mostrará el nombre del mes, y en caso de que el usuario ingrese un número no válido, como 20, se mostrará el siguiente mensaje: *Mes no válido*.

Si requieres usar un programa donde las comparaciones sean muy específicas, puedes emplear un `switch`; en el caso de evaluar rangos, utiliza un `if`.



### 2.5. Estructuras de control cíclicas

Las estructuras de control cíclicas son bloques de instrucciones que se repiten un número de veces mientras se cumple, o hasta que se cumpla una condición (García, Rodríguez, Mingo, Imaz, Brazález, Larzabal et ál., 2000).

Como regla general puede decirse que se utilizará:

- Ciclo *for*, cuando se conozca de antemano el número exacto de veces que ha de repetirse un determinado bloque de instrucciones.
- Ciclo *do-while*, cuando no se conoce exactamente el número de veces que se ejecutará, pero se sabe que por lo menos se ha de ejecutar una.
- Ciclo *while-do*, cuando es posible que no deba ejecutarse ninguna vez.

Estas reglas son generales, y algunos programadores se sienten más cómodos utilizando principalmente una de ellas. Con mayor o menor esfuerzo, puede utilizarse cualquiera de ellas indistintamente.

#### 2.5.1. While

El ciclo *while* tiene como característica que primero evalúa la condición y solo si se cumple realiza las instrucciones, así es que si la condición no es cumplida desde un inicio, el control del programa no entrará en el bloque de instrucciones que se encuentran dentro del ciclo. Su declaración es como se muestra a continuación:

```
while (condicion){  
    Bloque de Instrucciones a repetir  
}
```

La condición que se dé, deberá ser evaluada *verdadera* para que el control del programa ingrese al bloque de instrucciones, y deberá llegar en algún punto a que esa misma condición se evalúe *falsa* y salga del bloque repetitivo. Si planteas mal la condición y ésta nunca se evalúa como *falsa*, el programa se repetirá infinitamente, lo que provocará un problema en la ejecución.



Observa el siguiente ejemplo.

```
package ejemplos;
import javax.swing.JOptionPane;
/**
 * @author ESAD
 */
public class ejemploWhile {
    public static void main(String[] args) {
        int n= Integer.parseInt(JOptionPane.showInputDialog("Ingresa el
9"));
        while(n != 9){
            n= Integer.parseInt(JOptionPane.showInputDialog("Ingresa el 9"));
        }
    }
}
```

En este ejemplo se pide al usuario ingresar el número 9, y el ciclo seguirá mientras **no** lo ingresen, en el momento en que la condición se deja de cumplir (precisamente al ingresar dicho número) el ciclo terminará y el programa también.

### 2.5.2. Do-while

Este ciclo permite la ejecución del bloque repetitivo al menos una vez, se cumpla o no la condición, pues ejecuta primero el bloque *do* y al finalizarlo evalúa la condición, si ésta se cumple regresa a ejecutar nuevamente el bloque *do*, de lo contrario termina la ejecución.

```
do{
    Bloque de instrucciones a repetir
}
While(condicion);
```

Veamos el siguiente ejemplo:

```
package ejemplos;
```



```
import javax.swing.JOptionPane;
/**
 * @author ESAD
 */
public class ejemploDoWhile {
    public static void main(String[] args) {
        int n;
        do{
            n= Integer.parseInt(JOptionPane.showInputDialog("Ingresa el 9"));
        }
        while(n != 9);
    }
}
```

Como se puede observar, el ciclo *while* es muy parecido al *do-while*, pero recordemos que el *while*, si desde un principio no se cumple la condición no entra al ciclo, y el *do-while* primero entra y después evalúa.

Compara los dos ejemplos, y observarás que en el ciclo *do-while* la petición del número la hacemos dentro del bloque que conforma el *do*, pues se tiene la certeza de que entrará a ejecutar esta porción del programa, al menos una vez. Mientras que en el ciclo *while* la petición se hace fuera del bloque de instrucciones que lo conforman, pues la evaluación se realiza antes de entrar al ciclo.

### 2.5.3. For

Este ciclo es especialmente útil cuando, de antemano, conoces el número de veces que quieres que se ejecute el mismo bloque de instrucciones, su declaración es como sigue:

```
for(inicio ; condición; incremento){
    Bloque de instrucciones repetitivas
}
```

**Inicio:** se debe colocar un valor numérico donde comenzará el conteo de ejecuciones.



**Condición:** se coloca la condición a evaluar en cada iteración del ciclo para que se siga ejecutando el número de veces que se cumpla esta condición, pues el momento en que se evalúe como falsa el ciclo dejará de repetirse.

**Incremento:** esta instrucción se evalúa como si fuese la última del bloque de instrucciones, e indica el incremento o decremento de la variable numérica de inicio, para reevaluar la condición en cada iteración.

Se expone a continuación el ejemplo:

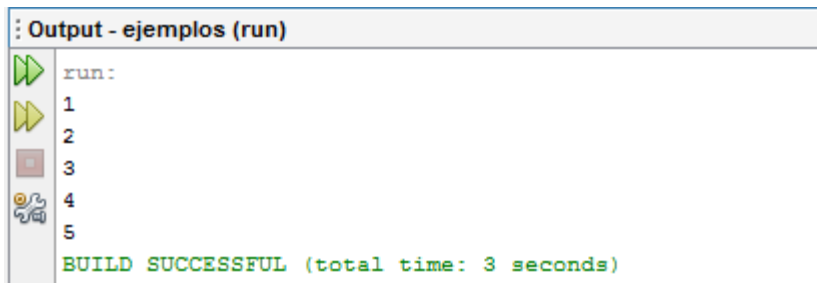
```
package ejemplos;

import javax.swing.JOptionPane;

/**
 * @author ESAD
 */

public class ejemploFor {
    public static void main(String[] args) {
        int n= Integer.parseInt(JOptionPane.showInputDialog("Cuántos
números quieres ver"));
        for(int i=1; i<=n; i=i+1){
            System.out.print(i);
        }
    }
}
```

Aquí se pide al usuario un número, y muestra esa misma cantidad de elementos. Veamos, el inicio empieza en 1, la condición para seguir es que la variable *i* sea menor o igual que el número dado por el usuario, y el incremento será de uno en uno; de esta manera, la salida del programa al ingresar un 5 sería como se muestra a continuación.



```
Output - ejemplos (run)

run:
1
2
3
4
5
BUILD SUCCESSFUL (total time: 3 seconds)
```





Otro punto importante respecto al ciclo *for* es que se puede anidar. Ésto se refiere a que puedo utilizar un ciclo dentro de otro, a manera de explicación observa el siguiente código y su salida.

```
package ejemplos;
import javax.swing.JOptionPane;
/**
 * @author ESAD
 */
public class forAnidado {
    public static void main(String[] args) {
        int n= Integer.parseInt(JOptionPane.showInputDialog("Cuantos
temas son"));
        int m= Integer.parseInt(JOptionPane.showInputDialog("Cuantos
subtemas contiene cada tema"));
        for(int i=1; i<=n; i=i+1){
            for(int j=1; j<=m; j=j+1){
                System.out.print(i+", "+j);
                System.out.print("\n");
            }
        }
    }
}
```

Como se puede ver, se tiene un ciclo dentro de otro. Su manera de funcionar es que entra al primero, después al segundo y, hasta que el segundo termine sus repeticiones, el primero volverá a incrementar su valor y volverá a entrar al segundo tomando sus valores iniciales. Veamos la salida. En este ejemplo se indicó que los temas eran tres, mientras que los subtemas dos, entonces se coloca el primer tema y sus respectivos subtemas 1.1.



y 1.2., ya que termina ese tema se sigue con el 2.1. y 2.2.

```
Output - ejemplos (run)
run:
1,1
1,2
2,1
2,2
3,1
3,2
BUILD SUCCESSFUL (total time: 10 seconds)
```

Nota: Si se quisieran anidar más de dos ciclos *for* es posible realizarlo, tomando en cuenta que siempre el ciclo que se encuentre más adentro será el que se realice primero.

También puede ser que el ciclo, en lugar de ir de manera ascendente, se realice de forma descendente, para esto se debe tomar en cuenta que el inicio sea el último elemento, la condición de paro el 0 y su incremento sea restar 1. Observa el siguiente ejemplo, que es el mismo que el ejemplo del *for*, pero ahora con un recorrido inverso.

```
package ejemplos;

import javax.swing.JOptionPane;

/**
 * @author ESAD
 */

public class ejemploFor2 {
    public static void main(String[] args) {
        int n= Integer.parseInt(JOptionPane.showInputDialog("Cuántos números quieres ver"));
        for(int i=n; i>0; i=i-1){
            System.out.print(i);
            System.out.print("\n");
        }
    }
}
```



```
Output - ejemplos (run)
run:
5
4
3
2
1
BUILD SUCCESSFUL (total time: 6 seconds)
```

Al realizar los cambios comentados, podemos observar que la salida nos muestra los cinco números, pero de manera descendente.

### Cierre de la unidad

Has concluido la unidad 2 del curso. A lo largo de ésta has visto lo que es una variable, su ámbito y declaración, así como las generalidades de los métodos, sus características, llamadas y tipos de parámetros que se manejan. Posteriormente, te involucraste conociendo las estructuras de control selectivas, como lo son el *If-else*, *If* anidado y *switch-case*.

Al final de la unidad, también se vieron las estructuras de control cíclicas *while*, *do-while* y *for*, cuyo propósito ha sido que obtengas conocimientos sobre estructuras de control, y logres aplicarlas en la resolución de problemas y creación de programas para dichas soluciones.

Es aconsejable que revises nuevamente la unidad en caso de que los temas que se acaban de mencionar no te sean familiares o no los recuerdes, de no ser este tu caso, ya estás preparado(a) para seguir con la unidad 3, en donde continuarás con la revisión de herencia, polimorfismo y excepciones en un programa Java. Todo ello con el fin de obtener el conocimiento necesario para comenzar a realizar pequeños programas computacionales al final de la cuarta y última unidad del curso de Programación orientada a objetos I.



### Para saber más

Consulta la página oficial del lenguaje Java: <http://www.java.com/es/>, donde podrás encontrar manuales de referencia sobre los métodos, variables, estructuras de control selectivas y de repetición.

### Fuentes de consulta

- García, J., Rodríguez, J., Mingo, I., Imaz, A., Brazález, A., Larzabal, A. et ál. (2000). *Aprenda Java como si estuviera en primero*. Recuperado de <http://www.tecnun.es/asignaturas/Informat1/AyudaInf/aprendainf/Java/Java2.pdf>
- Gosling, J., Holmes, D. y Arnold, K. (2001). *El lenguaje de programación Java TM*. Recuperado de [http://icesecurity.org/programacion/JAVA/Libro\\_Java-SP.pdf](http://icesecurity.org/programacion/JAVA/Libro_Java-SP.pdf)