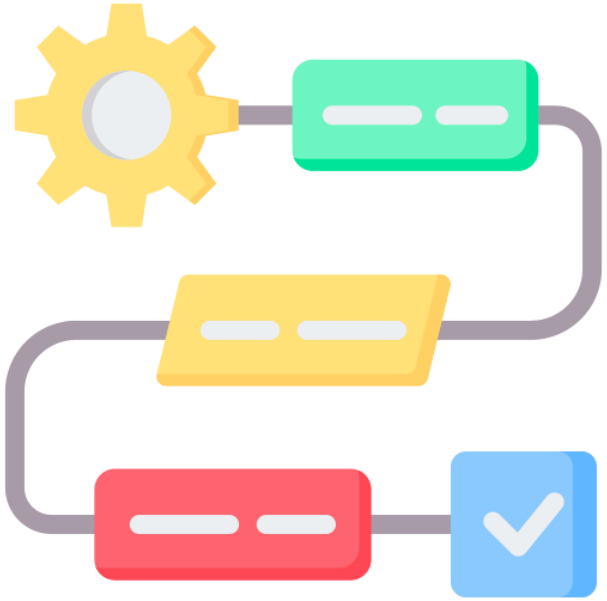
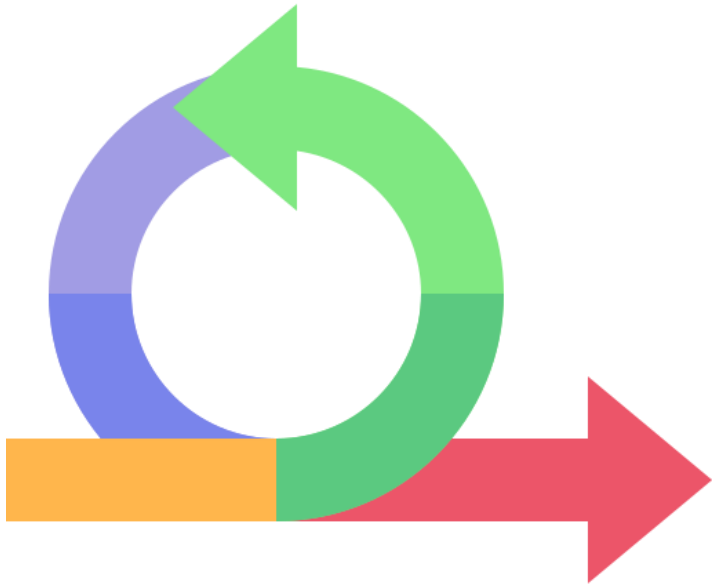
U04 – Condicionales, Bucles y Métodos

Condicionales



Métodos



Bucles

Unidad 04

❑ 4.1 Condicionales

Objetivos

- Escribir y probar código que haga uso de estructuras de selección.
- Clasificar, reconocer y utilizar los operadores del lenguaje en expresiones.
- Reconocer la estructura de un programa informático, identificando y relacionando los elementos propios de Java.
- Diseñar aplicaciones cuya ejecución no es siempre lineal, permitiendo, a partir de la entrada de datos, discriminar entre distintos escenarios.

Contenidos

- 2.1. Expresiones lógicas
- 2.2. Condicional simple: if
- 2.3. Condicional doble: if-else
- 2.4. Condicional múltiple: switch

❑ 4.2 Bucles

Objetivos

- Conocer y utilizar las estructuras de repetición.
- Programar aplicaciones que repiten conjuntos de instrucciones mediante el uso de bucles.
- Distinguir entre un bucle controlado por contador, un bucle precondición y un bucle poscondición.
- Utilizar las estructuras adecuadas de control para conseguir que un programa funcione según las especificaciones funcionales.

Contenidos

- 3.1. Bucles controlados por condición
- 3.2. Bucles controlados por contador: for
- 3.3. Salidas anticipadas
- 3.4. Bucles anidados

❑ 4.3 Métodos

Objetivos

- Asimilar el concepto de función, las ventajas de su uso y la implicación en la mejora del mantenimiento de aplicaciones.
- Entender y usar el concepto de parámetro de entrada, así como el mecanismo para generalizar el comportamiento de las funciones.
- Escribir programas que hagan un uso adecuado de las funciones y del valor devuelto por estas.
- Resolver problemas mediante el uso de funciones recursivas.

Contenidos

- 4.1. Conceptos básicos
- 4.2. Ámbito de las variables
- 4.3. Paso de información a una función
- 4.4. Valor devuelto por una función
- 4.5. Sobrecarga de funciones
- 4.6. Recursividad

❑ 4.1 Condicionales

❑ 4.1 Condicionales

Objetivos

- Escribir y probar código que haga uso de estructuras de selección.
- Clasificar, reconocer y utilizar los operadores del lenguaje en expresiones.
- Reconocer la estructura de un programa informático, identificando y relacionando los elementos propios de Java.
- Diseñar aplicaciones cuya ejecución no es siempre lineal, permitiendo, a partir de la entrada de datos, discriminar entre distintos escenarios.

Contenidos

- 2.1. Expresiones lógicas
- 2.2. Condicional simple: if
- 2.3. Condicional doble: if-else
- 2.4. Condicional múltiple: switch

❑ 4.1 Condicionales - Expresiones lógicas

Símbolo	Descripción
==	Igual que
!=	Distinto que
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que

Tabla 2.1. Operadores relacionales

Símbolo	Descripción
&&	Operador Y
||	Operador O
!	Operador negación

Tabla 2.2. Operadores lógicos

4.1 Condicionales - Expresiones lógicas

Realiza un programa que almacene la evaluación de distintas expresiones relacionales en variables booleanas, y muestra el valor de dichas variables.

```
package codigo;

public class Main {

    public static void main(String[] args) {
        boolean dosMayorTres = 2>3;
        boolean tresMayorDos = 3>2;
        boolean unoIgualUno = 1==1;

        System.out.println("Dos es mayor que tres: " + dosMayorTres);
        System.out.println("Tres es mayor que dos: " + tresMayorDos);
        System.out.println("Uno es igual que uno: " + unoIgualUno);
    }
}
```

❑ 4.1 Condicionales - Expresiones lógicas

Solicita por teclado un número de tipo `int` al usuario y escribe un programa que muestre `true` o `false`, dependiendo de si el número es positivo.

```java
package codigo;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        System.out.println("Escriba un número: ");
        Scanner sc = new Scanner(System.in);
        int numero = sc.nextInt();
        boolean esPositivo = numero > 0;
        System.out.println("Es " + numero + " positivo? " + esPositivo);
    }
}
```

❑ 4.1 Condicionales - Expresiones lógicas

Tabla 2.3. Tabla de verdad del operador **&&**

a	b	a && b
falso	falso	falso
cierto	falso	falso
falso	cierto	falso
cierto	cierto	cierto

Tabla 2.4. Tabla de verdad del operador **||**

a	b	a || b
falso	falso	falso
cierto	falso	cierto
falso	cierto	cierto
cierto	cierto	cierto

Tabla 2.5. Tabla de verdad del operador **!**

a	!a
falso	cierto
cierto	falso

❑ 4.1 Condicionales - Expresiones lógicas

a + b <= 18 (será true si el valor de a más el valor de b es menor o igual que 18)

¿devuelve `true` o `false`?

`a = 10 y b = 2`

`a = 20 y b = 1`

`2 x 5 == 10`

`1 != 2`

❑ 4.1 Condicionales - Expresiones lógicas

- Supongamos que **a vale 3** y **b vale 5**

¿devuelve `true` o `false`?

```
!(a + b <= 18)

(a + b <= 18) && (a == 4)

!(a + b <= 18) || (a == 4)

(a + b <= 18) || (cualquier condición)

(a == 4) && (cualquier condición)
```

4.1 Condicionales - Expresiones lógicas

Escribe una aplicación que pida al usuario dos números enteros y muestre: `true`, si ambos números son distintos entre sí o alguno de ellos es cero; y `false` en caso contrario.

```java
package codigo;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Escriba un número:");
        int num1 = sc.nextInt();
        System.out.println("Escriba otro número:");
        int num2 = sc.nextInt();

        boolean sonDistintosOAlgunoCero = num1 != num2 || num1 == 0 || num2 ==0;
        System.out.println("Son distintos o alguno es 0: " + sonDistintosOAlgunoCero);
    }

}
```

4.1 Condicionales - Expresiones lógicas

Realiza un programa que informe al usuario (mostrando `true`) si un primer número es múltiplo de otro número. Ambos números se piden por teclado.

```java
package codigo;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Escriba un número:");
        int num1 = sc.nextInt();
        System.out.println("Escriba otro número:");
        int num2 = sc.nextInt();

        boolean multiplos = num1 % num2 == 0;
        System.out.println(num1 + " es multiplo de " + num2 + ": " + multiplos);
    }
}
```

4.1 Condicionales - Condicional simple: if

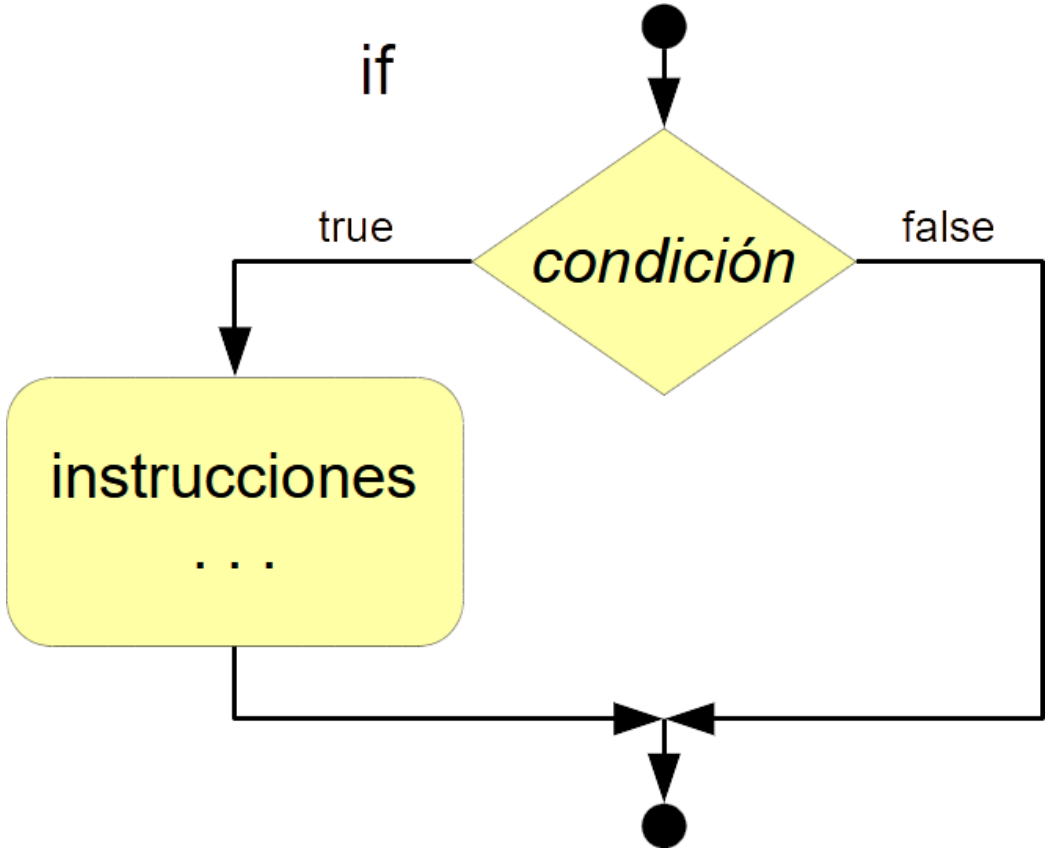


Diagrama de flujo de la instrucción **if**.

- Si la evaluación de la condición es **true**, se ejecutará el bloque de instrucciones que contiene **if**. Este bloque de instrucciones puede albergar cualquier tipo de sentencia.

```
if (condición) {
   bloque de instrucciones
   ...
}
```

Sintaxis en java de la instrucción **if**.

❑ 4.1 Condicionales - Condicional simple: if

¿ Qué escribe por pantalla ?

```
a = 3;
if (a + 1 < 10) {
    a = 0;
    System.out.println("Hola");
}
System.out.println("El valor de a es: " + a);
```

- El lugar o bloque donde es posible utilizar una variable se denomina **ámbito de la variable**.
- Hasta ahora hemos visto dos ámbitos de variables: las variables declaradas en main, que se denominan **variables locales**, y las variables declaradas en un bloque de instrucción, denominadas **variables de bloque**.

❑ 4.1 Condicionales - Condicional simple: if

¿ Qué escribe por pantalla ?

```
a = 9;
if (a + 1 < 10) {
   a = 0;
   System.out.println("Hola");
}
System.out.println ("El valor de a es: " + a);
```

❑ 4.1 Condicionales - Condicional doble: if - else

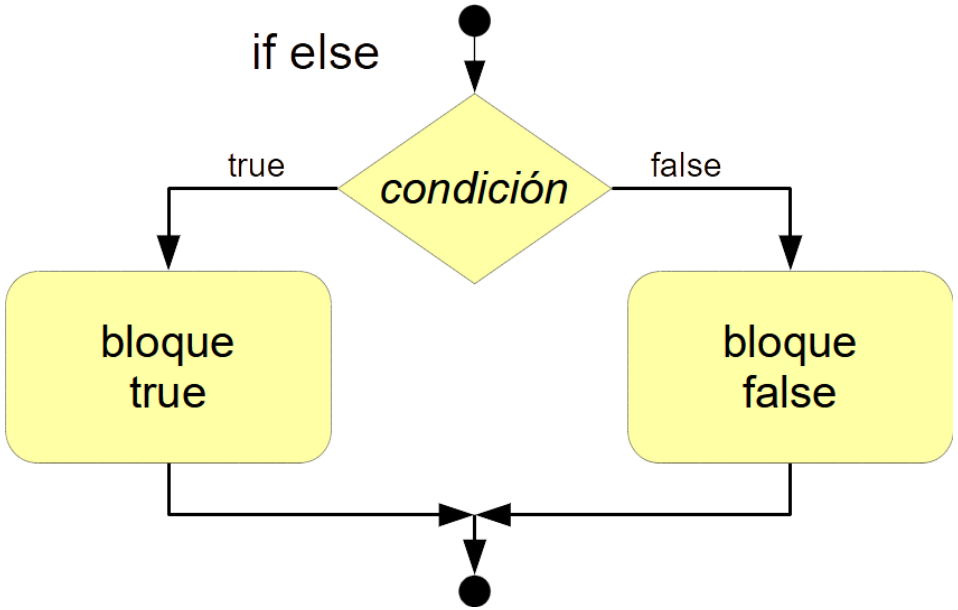


Diagrama de flujo de la instrucción **if-else**.

- El bloque **true** se ejecutará cuando la condición resulte verdadera y el bloque **false** cuando la condición resulte falsa.

```
if (condición) {
   bloque true //se ejecuta cuando la condición es cierta
} else {
   bloque false //se ejecuta cuando la condición es falsa
}
```

Sintaxis en java de la instrucción **if-else**.

❑ 4.1 Condicionales - Condicional doble: if - else

- El **operador ternario** permite seleccionar un valor de entre dos posibles, dependiendo de la evaluación de una condición. La sentencia

```
variable = condición ? valor1 : valor2
```

- Es equivalente a utilizar un condicional doble de la forma

```
if (condición) {
   variable = valor1;
} else {
   variable = valor2;
}
```

❑ 4.1 Condicionales - Condicional doble: if - else

- Es recomendable utilizar el **operador ternario** por economía y legibilidad del código, en lugar de un **if-else**, cuando sea posible.

- Un **ejemplo** para calcular el máximo de dos números introducidos por teclado es:

```
Scanner sc = new Scanner(System.in);
int a = sc.nextInt();
int b = sc.nextInt();
int maximo = a > b ? a : b;
System.out.println("El máximo es: " + maximo);
```

4.1 Condicionales - Condicional doble: Anidación de condicionales

```
if (a - 2 == 1) {
   System.out.println("Hola ");
} else {
   if (a - 2 == 5) {
      System.out.println("Me ");
   } else {
      if (a - 2 == 8) {
         System.out.println("Alegro ");
      } else {
         if (a - 2 == 9) {
            System.out.println("De ");
         } else {
            if (a - 2 == 11) {
               System.out.println("Conocerte.");
            } else {
               System.out.println("Sin coincidencia");
            }
         }
      }
   }
}
```

❑ 4.1 Condicionales - Condicional doble: Anidación de condicionales

- De todas formas, cuando hay muchos casos alternativos es habitual **eliminar las llaves de los bloques else**, consiguiendo un código más compacto.

- De esta manera, el ejemplo anterior podría reescribirse:

```
if (a - 2 == 1) {
   System.out.println("Hola ");
} else if (a - 2 == 5) {
   System.out.println("Me ");
} else if (a - 2 == 8) {
   System.out.println("Alegro ");
} else if (a - 2 == 9) {
   System.out.println("De ");
} else if (a - 2 == 11) {
   System.out.println("Conocerte.");
} else {
   System.out.println("Sin coincidencia");
}
```

❑ 4.1 Condicionales

ACTIVIDADES

1) **Diseñar una aplicación que solicite al usuario un número e indique si es par o impar.**

2) **Pedir dos números enteros y decir si son iguales o no.**

3) **Solicitar dos números distintos y mostrar cuál es el mayor.**

4) **Pedir dos números y mostrarlos ordenados de forma decreciente.**

5) **Solicitar dos números y mostrar cuál es el mayor, considerando el caso de que los números introducidos sean iguales.**

6) **Pedir tres números y mostrarlos ordenados de mayor a menor.**

7) **Escribir una aplicación que indique cuántas cifras tiene un número entero introducido por teclado, que estará comprendido entre 0 y 99.999.**

❑ 4.1 Condicionales - Condicional múltiple: switch

- En ocasiones, el hecho de utilizar **muchos if o if-else anidados** suele producir un código poco legible y difícil de mantener.

1. Evalúa **expresión** y obtiene su valor.
2. Compara, **uno a uno**, el valor obtenido con cada valor de las cláusulas case.
3. En el momento en que **coincide** con alguno de ellos, ejecuta el conjunto de instrucciones de esa cláusula case y de todas las siguientes.
4. Si **no existe coincidencia alguna**, se ejecuta el conjunto de instrucciones de la cláusula **default**, siempre y cuando esta esté presente.

```
switch (expresión) {
   case valor1:
      conjunto instrucción 1
   case valor2:
      conjunto instrucción 2
   ...

   case valorN:
      conjunto instrucción N
   default:
      conjunto instrucción default
}
```

Sintaxis en java de la instrucción **switch**.

4.1 Condicionales - Condicional múltiple: switch

- **break** impide que se ejecuten los siguientes bloques de instrucciones

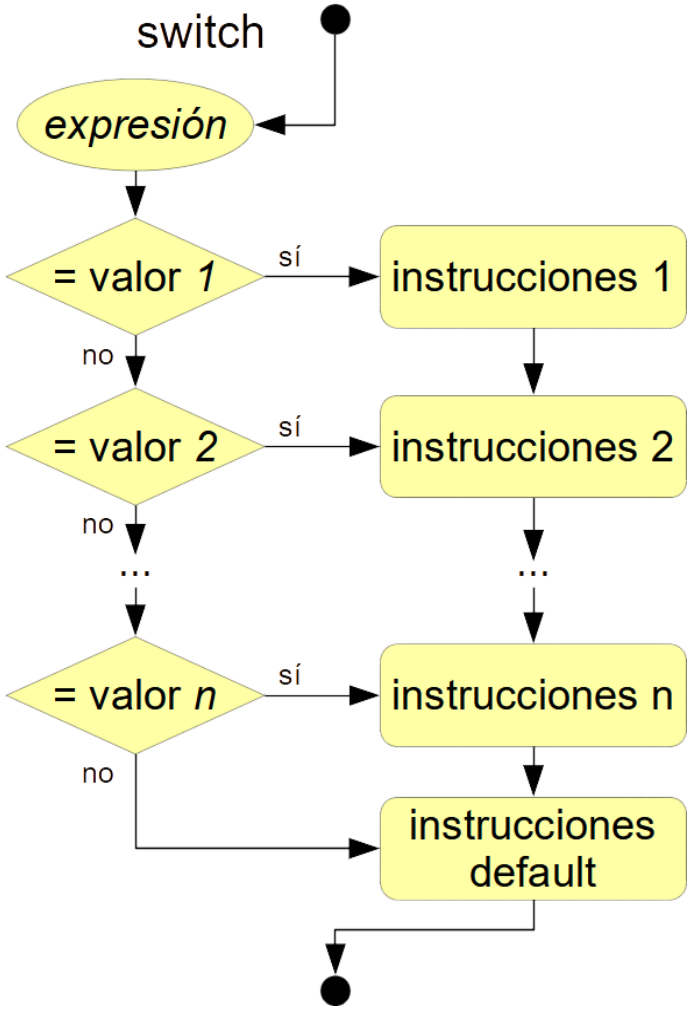


Diagrama de flujo de la instrucción **switch**.

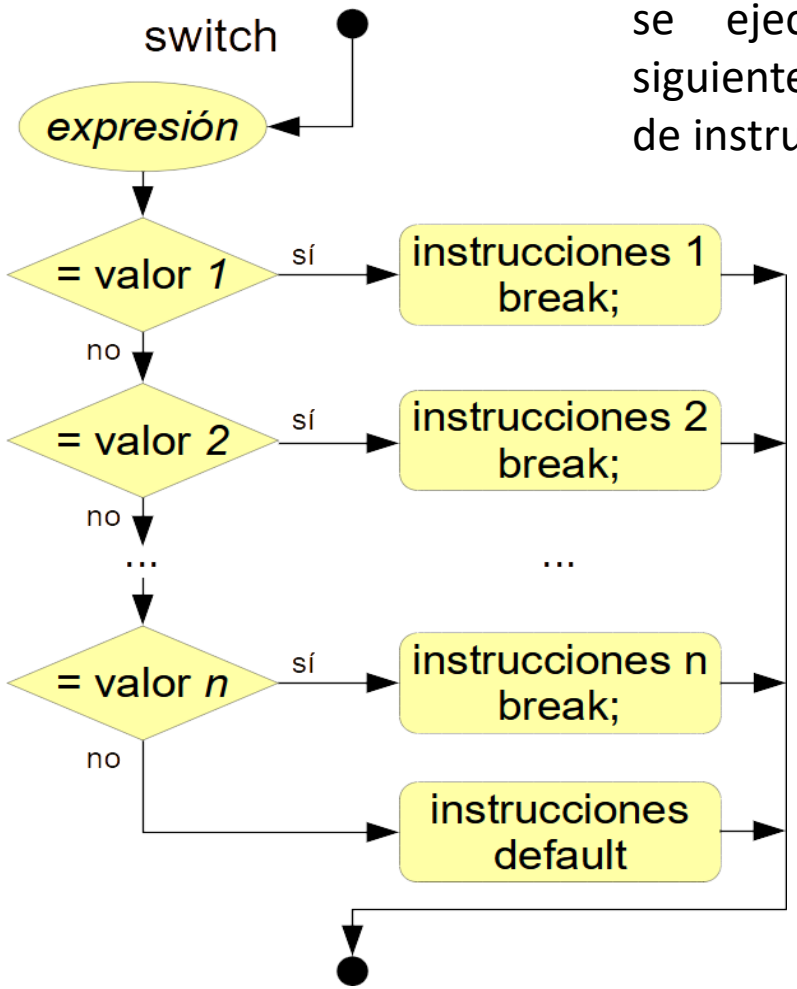


Diagrama de flujo de la instrucción **switch** con **break**.

❑ 4.1 Condicionales - Condicional múltiple: switch

```
a = 10;
switch (a-2) {
   case 1:
      System.out.print("Hola ");
   case 5:
      System.out.print("Me ");
   case 8:
      System.out.print("Alegro ");
   case 9:
      System.out.print("De ");
   case 11:
      System.out.print("Conocerte. ");
   default:
      System.out.print("Sin coincidencia");
}
```

¿ Qué escribe por pantalla ?

4.1 Condicionales - Condicional múltiple: switch

¿ Qué escribe por pantalla ?

```
a = 1;
switch (a*2) {
   case 1:
      System.out.println("Hola");
      break;
   case 2:
      System.out.println("Paco");
      break;
   case 3:
      System.out.println("Adiós");
      break;
   default:
      System.out.println("Sin coincidencia");
      break;
}
```

4.1 Condicionales - Condicional múltiple: switch

```
a = 1;
switch (a*2) {
   case 1:
      System.out.println("Hola");
      break;
   case 2:
      System.out.println("Paco");
   case 3:
      System.out.println("Adiós");
      break;
   default:
      System.out.println("Sin coincidencia");
}
```

¿ Qué escribe por pantalla ?

❑ 4.1 Condicionales - Condicional múltiple: switch

- En sucesivas nuevas versiones de Java (**v12**) se ha añadido una **nueva forma de utilizar la instrucción switch** en la que no es necesario usar break.

- Si el valor de la expresión coincide con algún case, solamente se ejecutará el bloque de instrucciones asociado a dicho case. La forma de distinguir entre la versión clásica de switch y la nueva es que esta, en lugar de utilizar dos puntos (`:`) después de cada case, emplea un guion seguido de un mayor que (`->`).

- En la nueva versión de switch, cada bloque de instrucciones de un case debe ir entre llaves, excepto si el bloque de instrucciones está formado por una única instrucción.

- El **nuevo switch** también permite que se use como una expresión, es decir, **toda la sentencia switch se sustituirá por un valor**, con el que podremos operar o simplemente asignarlo a una variable. Para que switch se use como una expresión, dentro de cada bloque de instrucciones case debe especificarse el valor que sustituirá a la instrucción switch. Para ello usaremos la palabra reservada **`yield`**. El uso de **`yield`** implica que todos los bloques de instrucciones deberán estar delimitados por llaves.

https://medium.com/@javatechie/the-evolution-of-switch-statement-from-java-7-to-java-17-4b5eee8d29b7
https://softwaremill.com/java-21-switch-the-power-on/

4.1 Condicionales - Condicional múltiple: switch ->

```
switch (nota) {
   case 0,1,2,3,4 -> { //bloque formado por dos instrucciones: entre llaves
      System.out.println("Suspenso.");
      System.out.println("Ánimo...");
   }
   case 5 -> //bloque de una única instrucción: podemos obviar las llaves
      System.out.println("Suficiente.");
   case 6 ->
      System.out.println("Bien.");
   case 7, 8 ->
      System.out.println("Notable");
   case 9, 10 -> {
      System.out.println("Sobresaliente.");
      System.out.println("Enhorabuena");
   }
   default ->
      System.out.println("Nota incorrecta");
}
```

4.1 Condicionales - Condicional múltiple: switch `yield`

```
System.out.println("Escriba un mes (1 al 12):");
int mes = new Scanner(System.in).nextInt();
int dias = switch (mes) {
   case 1, 3, 5, 7, 8, 10, 12 -> {
      yield 31; //estos meses tienen 31 días
   }
   case 2 -> {
      yield 28; //febrero tiene 28 días
   }
   case 4, 6, 9, 11 -> {
      yield 30; //el resto de meses tiene 30 días
   }
   default -> {
      System.out.println("Error: el mes es incorrecto");
      yield -1; //con -1 indicamos que hay un error
   }
};
System.out.println("Días: " + dias);
```

- En este ejemplo hay que señalar que la **asignación** no finaliza hasta que se escribe toda la instrucción switch, y que, como toda instrucción, necesita su **punto y coma final**.
- Toda la sentencia switch se sustituye por el valor que, en cada caso, indique **`yield`**.

❑ 4.1 Condicionales - Condicional múltiple: switch

ACTIVIDADES

8) **Pedir una nota entera de 0 a 10 y mostrarla de la siguiente forma: insuficiente (de 0 a 4), suficiente (5), bien (6), notable (7 y 8) y sobresaliente (9 y 10).**

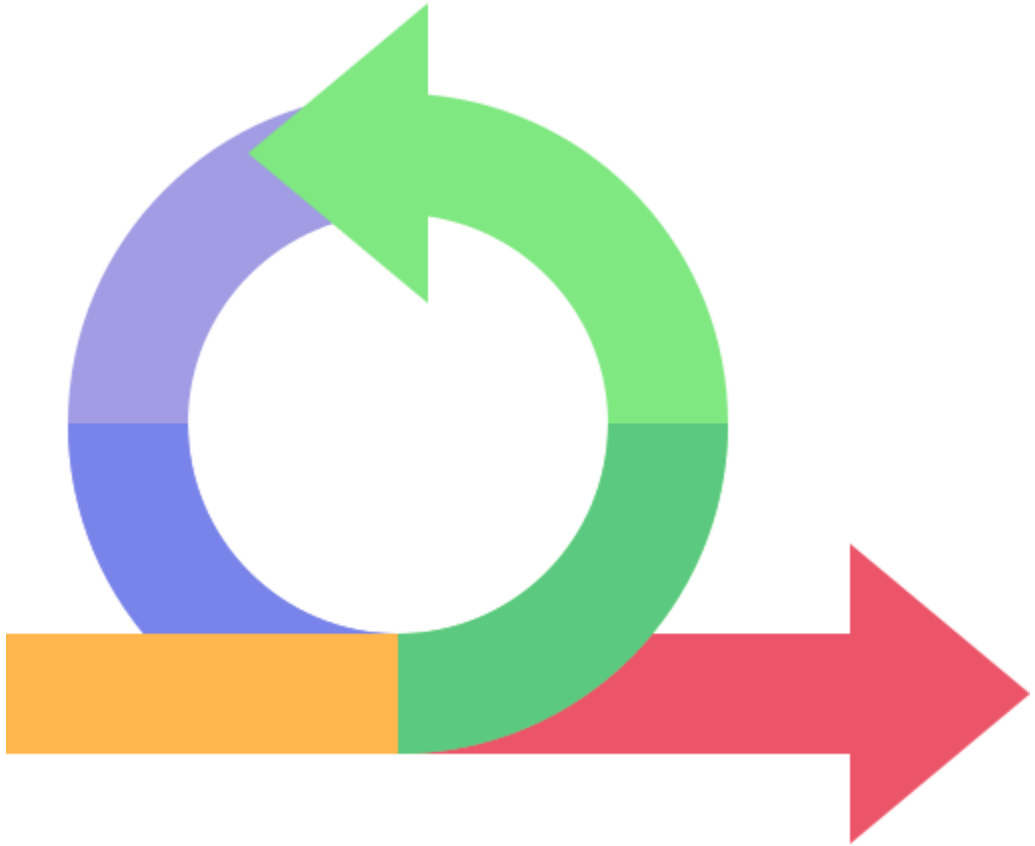
9) **Idear un programa que solicite al usuario un número comprendido entre 1 y 7, correspondiente a un día de la semana. Se debe mostrar el nombre del día de la semana al que corresponde. Por ejemplo, el número 1 corresponde a «lunes» y el 6 a «sábado».**

10) **Pedir el día, mes y año de una fecha e indicar si la fecha es correcta. Hay que tener en cuenta que existen meses con 28, 30 y 31 días (no se considerará los años bisiestos).**

11) **Escribir un programa que pida una hora de la siguiente forma: hora, minutos y segundos. El programa debe mostrar qué hora será un segundo más tarde. Por ejemplo: hora actual [10:41:59] hora actual +1 segundo: [10:42:00]**

12) **Crear una aplicación que solicite al usuario una fecha (día, mes y año) y muestre la fecha correspondiente al día siguiente.**

❑ 4.2 Bucles



Un **bucle** es un tipo de estructura que contiene un bloque de instrucciones que **se ejecuta repetidas veces**; cada ejecución o repetición del bucle se llama **iteración**.

❑ 4.2 Bucles

Objetivos

- Conocer y utilizar las estructuras de repetición.
- Programar aplicaciones que repiten conjuntos de instrucciones mediante el uso de bucles.
- Distinguir entre un bucle controlado por contador, un bucle precondición y un bucle poscondición.
- Utilizar las estructuras adecuadas de control para conseguir que un programa funcione según las especificaciones funcionales.

Contenidos

- 3.1. Bucles controlados por condición
- 3.2. Bucles controlados por contador: for
- 3.3. Salidas anticipadas
- 3.4. Bucles anidados

❑ 4.2.1 Bucles - Bucles controlados por condición - `while`

- Al igual que la instrucción **`if`**, el comportamiento de **`while`** depende de la evaluación de una condición.
- El bucle **`while`** decide si realizar una nueva iteración basándose en el valor de la condición.

1. Se evalúa **`condición`**.
2. Si la evaluación resulta **`true`**, se ejecuta el bloque de instrucciones.
3. Tras ejecutarse el bloque de **`instrucciones`**, se vuelve al primer punto.
4. Si, por el contrario, la condición es **`false`**, terminamos la ejecución del bucle.

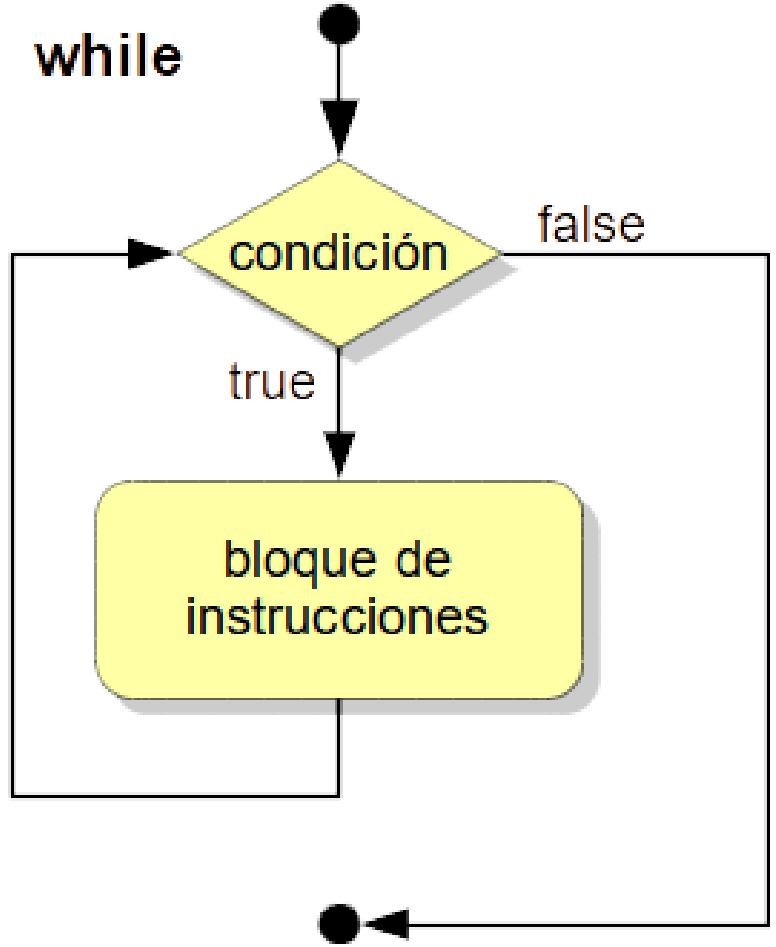


Diagrama de flujo de la instrucción **while**.

❑ 4.2.1 Bucles - Bucles controlados por condición - `while`

1. Se evalúa **`condición`**.
2. Si la evaluación resulta **`true`**, se ejecuta el bloque de instrucciones.
3. Tras ejecutarse el bloque de **`instrucciones`**, se vuelve al primer punto.
4. Si, por el contrario, la condición es **`false`**, terminamos la ejecución del bucle.

```
while (condición) {
    bloque de instrucciones
    ...
}
```

Sintaxis en java de la instrucción **while**.

❑ 4.2.1 Bucles - Bucles controlados por condición - `while`

- Por **ejemplo**, podemos mostrar los números del **1** al **3** mediante un bucle **`while`** controlado por la variable **`i`**, que empieza valiendo **1**, con la condición **`i <= 3`**:

```
int i = 1; //valor inicial
while (i <= 3) { //el bucle iterará mientras i sea menor o igual que 3
   System.out.println(i); //mostramos i
   i++; //incrementamos i
}
```

❑ 4.2.1 Bucles - Bucles controlados por condición - `while`

- Un bucle **`while`** puede realizar cualquier número de iteraciones, desde cero, cuando la primera evaluación de la condición resulta falsa, hasta infinitas, en el caso de que la condición sea siempre cierta.

- Esto es lo que se conoce como **bucle infinito**:

```
int cuentaAtras = 10;
while (cuentaAtras >= 0) {
    System.out.println(cuentaAtras);
}
```

- Dentro del bloque de instrucciones no hay nada que modifique la variable **`cuentaAtras`**, lo que hace que la condición permanezca idéntica, evaluándose siempre **`true`** y haciendo que el bucle sea **infinito**.

❑ 4.2.1 Bucles - Bucles controlados por condición - `while`

- En este caso, independientemente del bloque de instrucciones asociado a la estructura **`while`**, no se llega a entrar nunca en el bucle, debido a que la condición no se cumple ni siquiera la primera vez.
- Se realizan **cero iteraciones**.
- Veamos un **ejemplo** de un bucle **`while`** que **nunca llega a ejecutarse**:

```
int cuentaAtras = -8; //valor negativo
   while (cuentaAtras >= 0) {
   ...
}
```

4.2.1 Bucles - Bucles controlados por condición - `while`

```java
package codigo;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Escriba una edad: ");
        int edad = sc.nextInt();
        int edadMax = edad;
        int edadMin = edad;
        while (edad != -1) {
            if (edadMin > edad) {
                edadMin = edad;
            }
            if (edadMax < edad) {
                edadMax = edad;
            }
            System.out.print("Escriba una edad: ");
            edad = sc.nextInt();
        }

        System.out.println("La edad máxima es: " + edadMax);
        System.out.println("La edad mínima es: " + edadMin);
    }
}
```

- **Ejemplo: edad max y min**
  - Diseña una aplicación que muestre la edad máxima y mínima de un grupo de alumnos.
  - El usuario introducirá las edades y terminará escribiendo un -1.

❑ 4.2.1 Bucles - Bucles controlados por condición - `while`

```java
package codigo;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Escriba un número: ");
        int n = sc.nextInt();
        while (n > 0) {
            int unidad = n % 10;
            System.out.println(unidad);
            n /= 10;
        }
    }
}
```

- **Ejemplo: guarismos**
  - Implementa un programa que pida al usuario un número positivo y 10 muestre guarismo a guarismo.
  - Por ejemplo, para el número 123, debe mostrar primero el 3, a continuación el 2 y por último el 1.

4.2.1 Bucles - Bucles controlados por condición - do-while

- Disponemos de un segundo bucle controlado por una condición: el bucle **do-while**, muy similar a **while**, con la diferencia de que primero se ejecuta el bloque de instrucciones y después se evalúa la condición para decidir si se realiza una nueva iteración.

1. Se ejecuta el bloque de instrucciones.
2. Se evalúa condición.
3. Según el valor obtenido, se termina el bucle o se vuelve al punto 1.

```
do {
   bloque de instrucciones
   ...
} while (condición);
```

Sintaxis en java de la instrucción **do-while**.

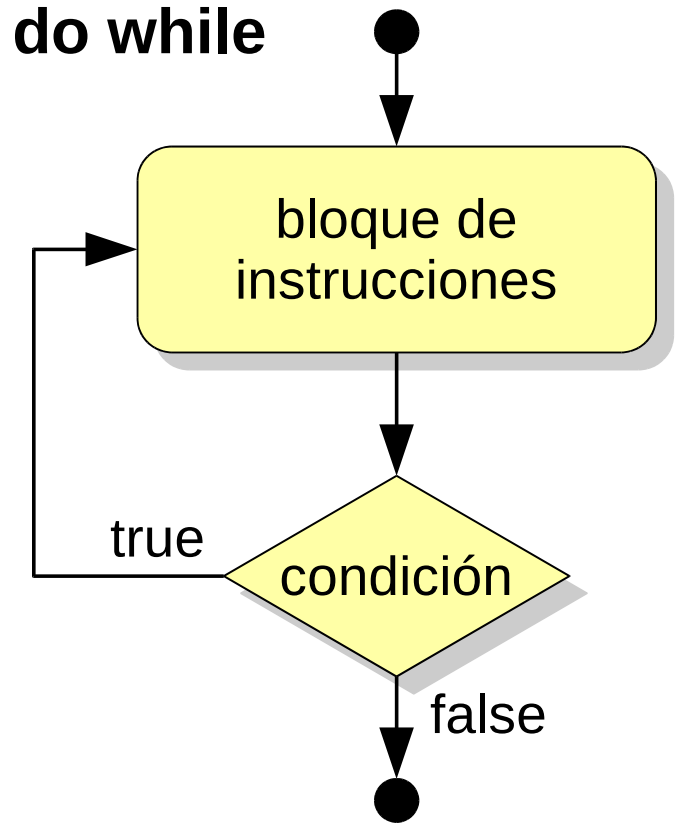


Diagrama de flujo de la instrucción **do-while**.

❑ 4.2.1 Bucles - Bucles controlados por condición - `do-while`

- Como ejemplo, vamos a escribir el código que muestra los números del 1 al 10 utilizando un bucle **`do-while`**, en vez de un bucle **`while`**:

```
int i = 1;
do {
   System.out.println(i);
   i++;
} while (i <= 10);
```

- Debemos recordar que es el único bucle que termina en punto y coma (**`;`**).
- Mientras el bucle **`while`** se puede ejecutar de **`0`** a infinitas veces, el **`do-while`** lo hace de **`1`** a infinitas veces.
- De hecho, la única diferencia con el bucle **`while`** es que **`do-while`** se ejecuta, al menos, una vez.

❑ 4.2.1 Bucles - Bucles controlados por condición - `while`

ACTIVIDADES
(while)

1) **Diseñar un programa que muestre, para cada número introducido por teclado, si es par, si es positivo y su cuadrado. El proceso se repetirá hasta que el número introducido sea 0.**

2) **Implementar una aplicación para calcular datos estadísticos de las edades de los alumnos de un centro educativo. Se introducirán datos hasta que uno de ellos sea negativo, y se mostrará: la suma de todas las edades introducidas, la media, el número de alumnos y cuántos son mayores de edad.**

❑ 4.2.1 Bucles - Bucles controlados por condición - `while`

ACTIVIDADES
(while)

3) Codificar el juego «el número secreto», que consiste en acertar un número entre 1 y 100 (generado aleatoriamente). Para ello se introduce por teclado una serie de números, para los que se indica: «mayor» o «menor», según sea mayor o menor con respecto al número secreto. El proceso termina cuando el usuario acierta o cuando se rinde (introduciendo un -1).

https://www.w3api.com/Java/Math/random/

4) Un centro de investigación de la flora urbana necesita una aplicación que muestre cuál es el árbol más alto. Para ello se introducirá por teclado la altura (en centímetros) de cada árbol (terminando la introducción de datos cuando se utilice -1 como altura). Los árboles se identifican mediante etiquetas con números únicos correlativos, comenzando en 0. Diseñar una aplicación que resuelva el problema planteado.

❑ 4.2.1 Bucles - Bucles controlados por condición - `do-while`

ACTIVIDADES
(do-while)

5) Desarrollar un juego que ayude a mejorar el cálculo mental de la suma. El jugador tendrá que introducir la solución de la suma de dos números aleatorios comprendidos entre 1 y 100. Mientras la solución introducida sea correcta, el juego continuará. En caso contrario, el programa terminará y mostrará el número de operaciones realizadas correctamente.

https://www.w3api.com/Java/Math/random/

❑ 4.2.2 Bucles - Bucles controlados por contador - for

- El bucle **for** permite controlar el número de iteraciones mediante una variable (que suele recibir el nombre de **contador**).

1. Se ejecuta la inicialización.
2. Se evalúa la condición: false, salimos del bucle; en caso de que la evaluación sea true, se ejecuta todo el bloque de instrucciones.
3. Se ejecuta el incremento.
4. Se vuelve de nuevo al punto 2.

```
for (inicialización; condición; incremento) {
   bloque de instrucciones
   ...
}
```

Sintaxis en java de la instrucción **for**.

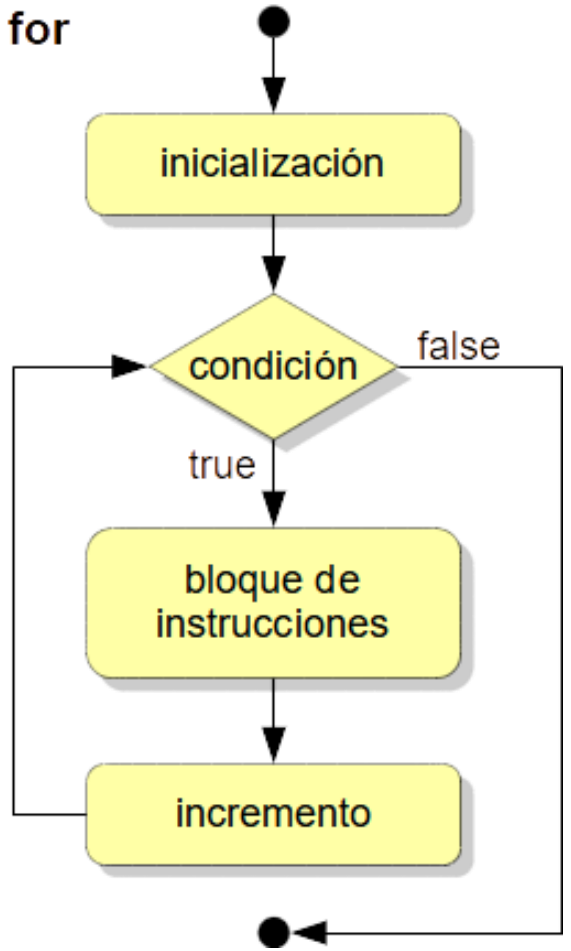


Diagrama de flujo de la instrucción **for**.

❑ 4.2.2 Bucles - Bucles controlados por contador - `for`

- Veamos un ejemplo donde solo se usa la variable **`i`** para controlar el bucle:

```
for (int i = 1; i <= 2; i++) {
   System.out.println("La i vale " + i);
}
```

¿ Qué imprime por pantalla ?

Argot técnico

En este caso, la variable `i`, además de inicializarse, también se declara en la zona de inicialización. Esto significa que `i` solo puede usarse dentro de la estructura `for`.

❑ 4.2.2 Bucles - Bucles controlados por contador - for

- **Ejemplo: Ecos**

  - Pide al usuario un número y muestra “Eco...” tantas veces como indique el número introducido.

  - Salida para el número 3:

```
Eco...
Eco...
Eco...
```

```java
package codigo;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Escriba una número: ");
        int veces = sc.nextInt();
        for (int i = 1; i <= veces; i++) {
            System.out.println("Eco...");
        }
    }
}
```

❑ 4.2.2 Bucles - Bucles controlados por contador - for

ACTIVIDADES (for)

6) **Escribir una aplicación para aprender a contar, que pedirá un número n y mostrará todos los números del 1 a n.**

7) **Escribir todos los múltiplos de 7 menores que 100.**

8) **Pedir diez números enteros por teclado y mostrar la media.**

9) **Implementar una aplicación que pida al usuario un número comprendido entre 1 y 10. Hay que mostrar la tabla de multiplicar de dicho número, asegurándose de que el número introducido se encuentra en el rango establecido.**

10) **Diseñar un programa que muestre la suma de los 10 primeros números impares.**

11) **Pedir un número y calcular su factorial. Por ejemplo, el factorial de 5 se denota 5! y es igual a 5x4x3x2x1 = 120.**

❑ 4.2.3 Salidas anticipadas – `break / continue`

- Dependiendo de la lógica que implementar en un programa, puede ser interesante **terminar un bucle antes de tiempo** y no esperar a que termine por su condición (realizando todas las iteraciones).

- Para poder hacer esto disponemos de:

  - **`break`**: interrumpe completamente la ejecución del bucle.

  - **`continue`**: detiene la iteración actual y continúa con la siguiente.

- Cualquier programa puede escribirse sin utilizar **`break`** ni **`continue`**; se recomienda evitarlos, ya que rompen la secuencia natural de las instrucciones.

- Veamos un ejemplo:

❑ 4.2.3 Salidas anticipadas – `break` / `continue`

- Veamos unos **ejemplos**:

```
i = 1;
while (i <= 10) {
   System.out.println("La i vale" + i);
   if (i == 2) {
      break;
   }
   i++;
}
```

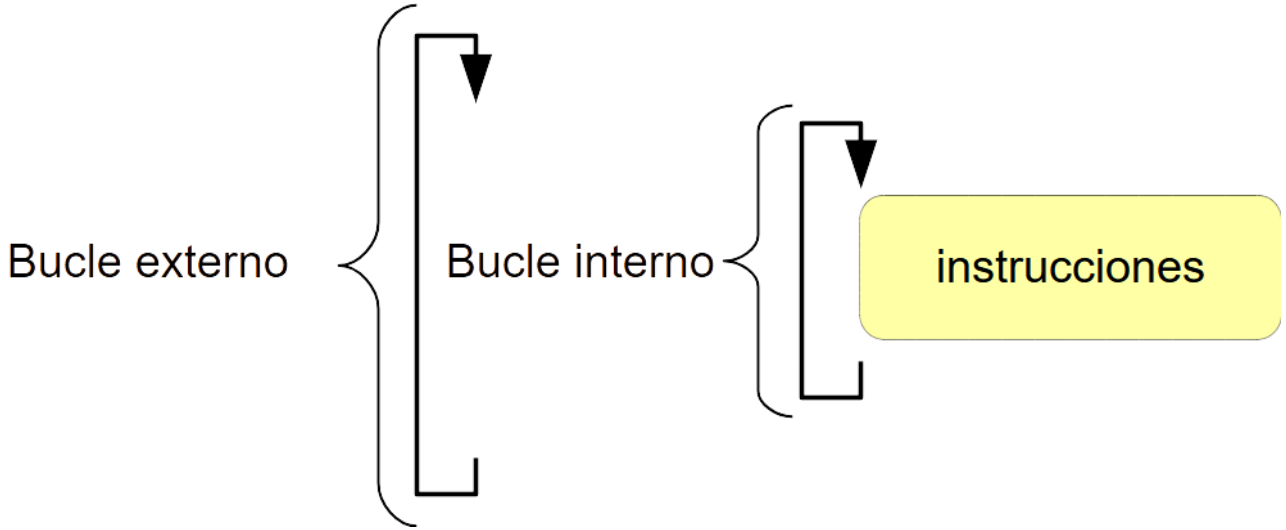
¿ Qué imprime por pantalla ?

¿ Qué imprime por pantalla ?

```
i = 0;
while (i < 10) {
   i++;
   if (i % 2 == 0) { //si i es par
      continue;
   }
   System.out.println("La i vale " + i);
}
```

❑ 4.2.4 Bucles anidados

- En el uso de los bucles es muy frecuente la **anidación**, que consiste en incluir un bucle dentro de otro.



- Al hacer esto se **multiplica** el número de veces que se ejecuta el bloque de instrucciones de los bucles internos.

- Los bucles anidados pueden encontrarse **dependientes** cuando las variables de los bucles más externos intervienen en el control de la iteración de un bucle interno; o **independientes**, cuando no existe relación alguna entre ellos.

4.2.4 Bucles anidados – Independientes

- Cuando los bucles anidados no dependen, en absoluto, unos de otros para determinar el número de iteraciones, se denominan **bucles anidados independientes**.

```
for (i = 1; i <= 4; i++) {
    for (j = 1; j <= 3; j++) {
        System.out.println("Ejecutando...");
    }
}
```

¿ Cuántas veces se ejecuta ?

❑ 4.2.4 Bucles anidados – Dependientes

- Son **bucles anidados dependientes** cuando el número de iteraciones de un bucle interno no sea independiente de la ejecución de los bucles exteriores, y dependa de sus variables de control.

```
for (i = 1; i <= 3; i++) {
   System.out.println("Bucle externo, i=" +i);
   j = 1;
   while (j <= i) {
      System.out.println("...Bucle interno, j=" + j);
      j++;
   }
}
```

¿ Qué imprime por pantalla ?

❑ 4.2.4 Bucles anidados

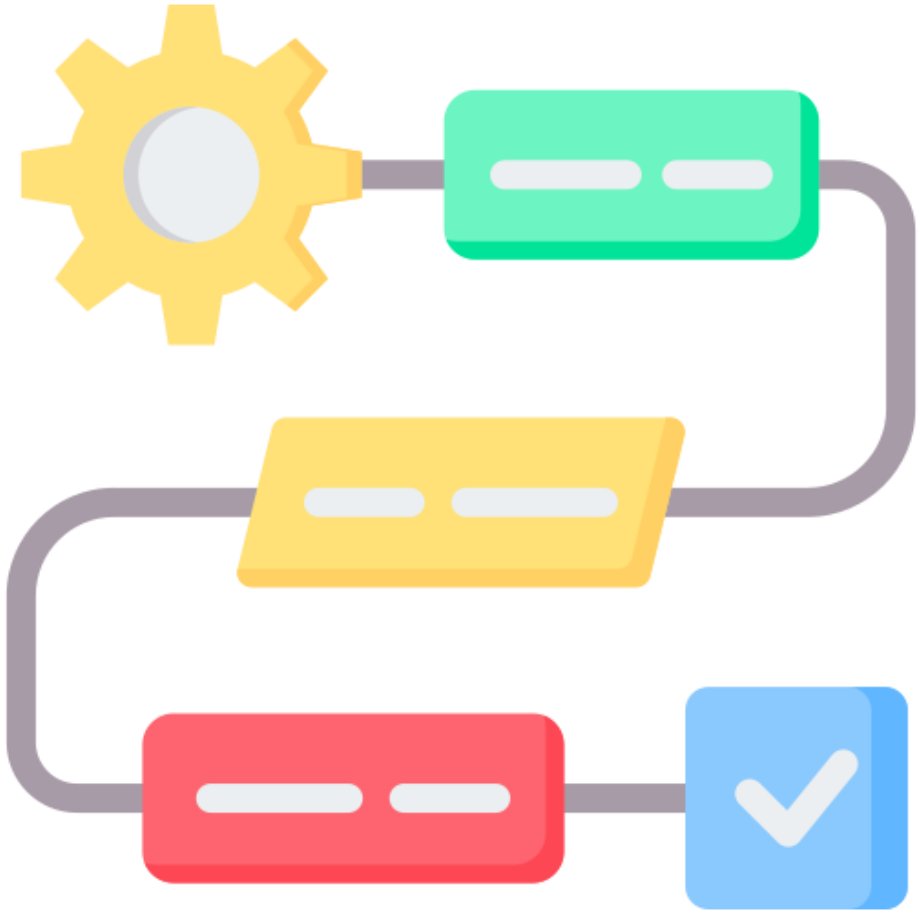
ACTIVIDADES
(anidados)

12) Diseñar una aplicación que muestre las tablas de multiplicar del 1 al 10.

13) Pedir por consola un número n y dibujar un triángulo rectángulo de n elementos de lado, utilizando para ello asteriscos (*). Por ejemplo, para n = 4:

```
* * * *
* * *
* *
*
```

❑ 4.3 Métodos



❑ 4.3 Métodos

Objetivos

- Asimilar el concepto de función, las ventajas de su uso y la implicación en la mejora del mantenimiento de aplicaciones.
- Entender y usar el concepto de parámetro de entrada, así como el mecanismo para generalizar el comportamiento de las funciones.
- Escribir programas que hagan un uso adecuado de las funciones y del valor devuelto por estas.
- Resolver problemas mediante el uso de funciones recursivas.

Contenidos

- 4.1. Conceptos básicos
- 4.2. Ámbito de las variables
- 4.3. Paso de información a una función
- 4.4. Valor devuelto por una función
- 4.5. Sobrecarga de funciones
- 4.6. Recursividad

❑ 4.3.1 Métodos – Conceptos básicos

- Conforme aumenta la **extensión** y la **complejidad** de un programa, es habitual tener que implementar, en distintas partes, la **misma funcionalidad**, cosa que implica copiar una y otra vez, donde sea necesario, el mismo fragmento de código.

- Esto genera dos **problemas**:

  - **Duplicidad del código**: aumenta el tamaño del código y lo hace menos legible.

  - **Dificultad en el mantenimiento**: cualquier modificación necesaria dentro del fragmento de código repetido tendría que realizarse en todos y cada uno de los lugares donde se encuentra.

❑ 4.3.1 Métodos – Conceptos básicos

```
public static void main(String[] args) {
    ... //código
    System.out.println("Voy a saludar tres veces:"); //fragmento repetido
    for(int i = 0; i < 3; i++) {
        System.out.println("Hola.");
    }
    ... //más código
    System.out.println("Voy a saludar tres veces:"); //fragmento repetido
    for(int i = 0; i < 3; i++) {
        System.out.println("Hola.");
    }
    ... //otro código
    System.out.println("Voy a saludar tres veces:"); //fragmento repetido
    for(int i = 0; i < 3; i++) {
        System.out.println("Hola.");
    }
    ... //resto del código
}
```

❑ 4.3.1 Métodos – Conceptos básicos

- La **solución** para cuando necesitamos la misma funcionalidad en distintos lugares de nuestro código no es más que etiquetar con un nombre un fragmento de código y sustituir en el programa dicho fragmento, en todos los lugares donde aparezca, por el nombre que le hemos asignado.

- Esta idea puede verse en el siguiente **ejemplo**:

```
public static void main(String[] args) {
    ... //código
    tresSaludos(); //sustitución por una función
    ... //más código
    tresSaludos(); //sustitución por una función
    ... //otro código
    tresSaludos(); //sustitución por una función
    ...//resto del código
}
```

❑ 4.3.1 Métodos – Conceptos básicos

- El **método** **`tressaludos()`** tendrá que definirse, de forma que se especifique el conjunto de instrucciones que la forma.

```
public static void main(String[] args) {
    ...
}
static void tresSaludos() {
    System.out.println("Voy a saludar tres veces:");
    for(int i = 0; i < 3; i++) {
        System.out.println("Hola.");
    }
}
```

¿Cómo se llama un método que no devuelve nada en programación estructurada?

- La **definición** de una función puede hacerse antes o después del **`main()`**.

❑ 4.3.1 Métodos – Conceptos básicos

- La **definición** de una función puede hacerse antes o después del **`main()`**.

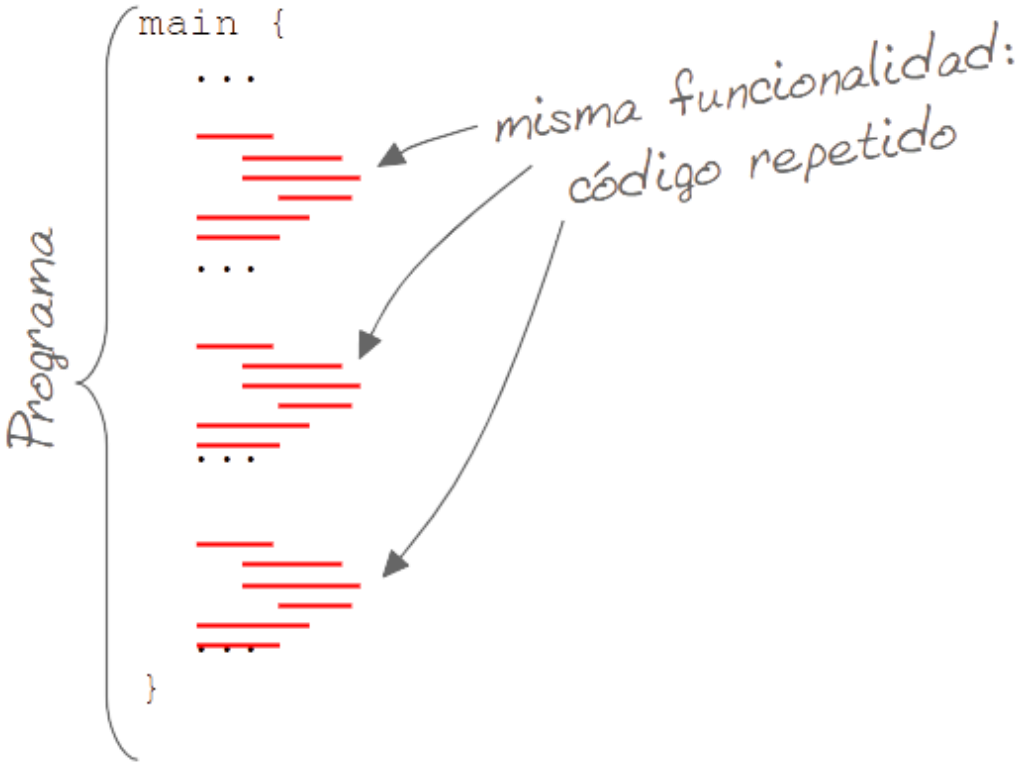
- Esto representa el concepto de **método**: un conjunto de instrucciones agrupadas bajo un nombre y con un objetivo común, que se ejecuta al ser invocada.

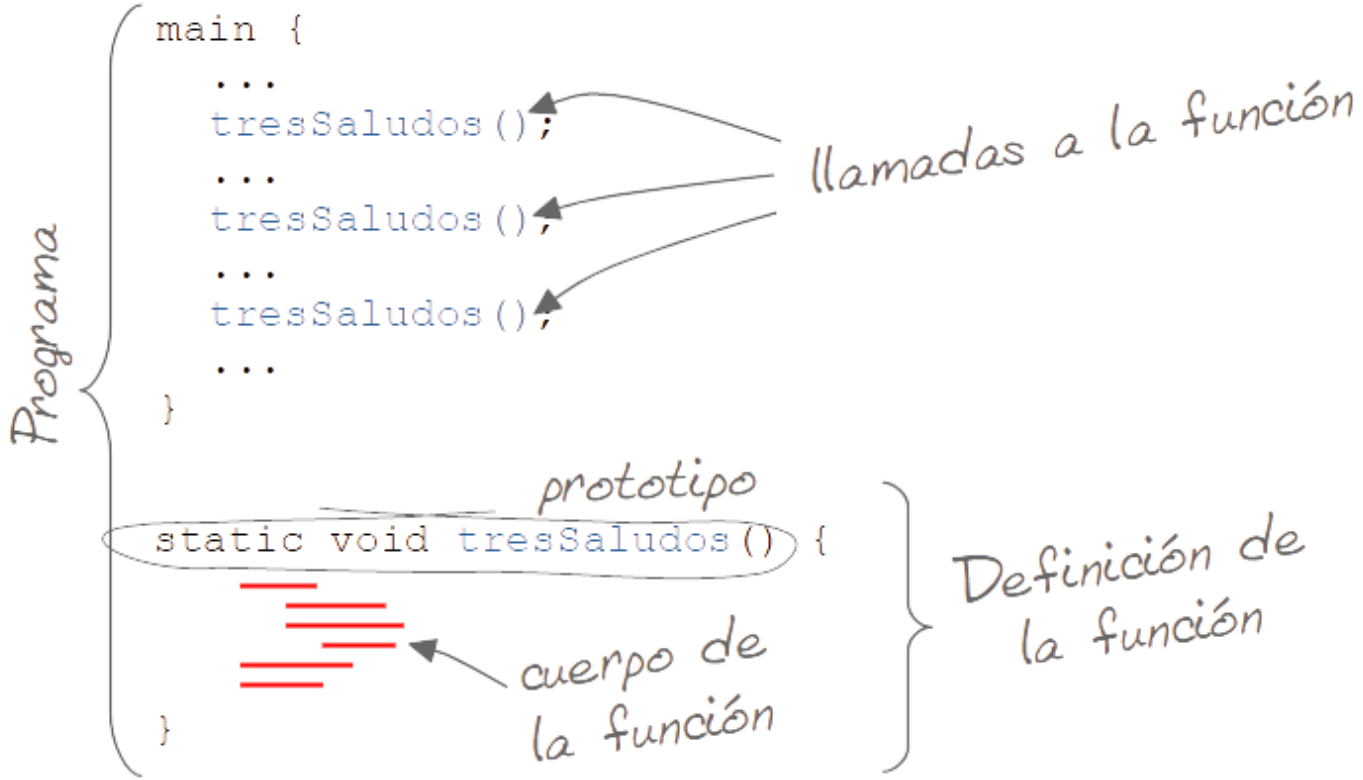
```
static tipo nombreFunción() {
    cuerpo de la función
}
```

Sintaxis en java de un **método**.

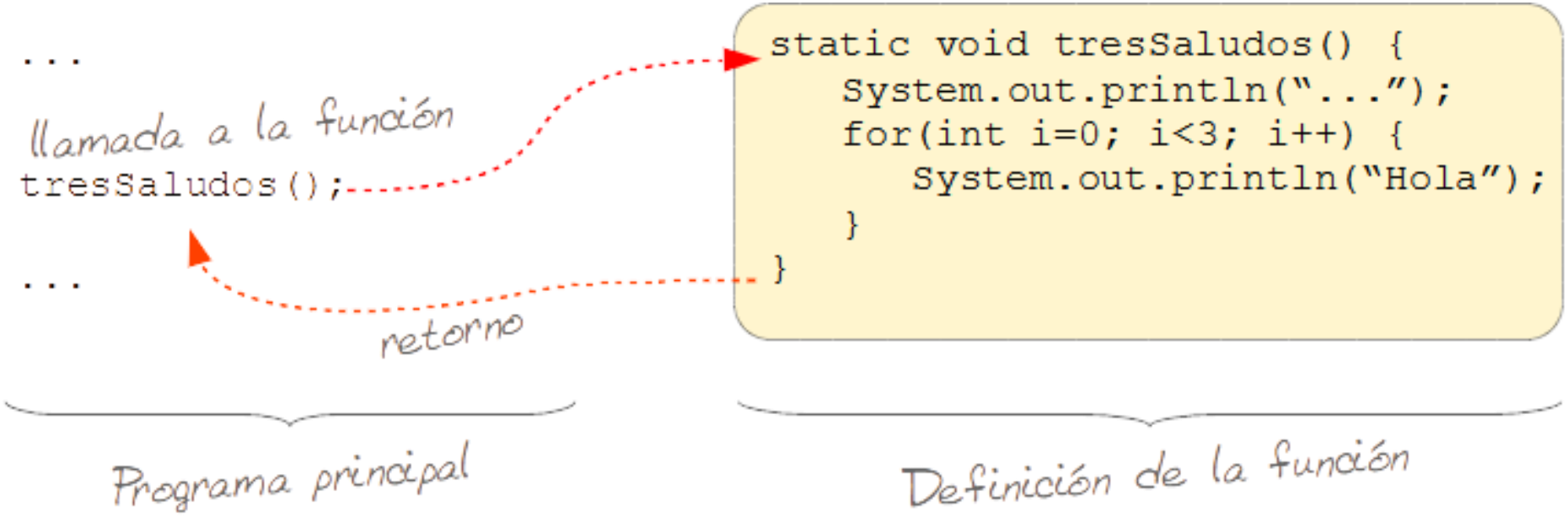
- En la **programación estructurada** se llaman **procedimientos** (si no devuelven ningún valor) o **funciones** (si devuelven valor) y en la **programación orientada a objetos**, se denominan **métodos**.

4.3.1 Métodos – Conceptos básicos





❑ 4.3.1 Métodos – Conceptos básicos



1. Las instrucciones del programa principal se ejecutan hasta que encuentra la llamada a la función.
2. La ejecución salta a la definición de la función.
3. Se ejecuta el cuerpo de la función.
4. Cuando la ejecución del cuerpo termina, retornamos al punto del programa desde donde se invoco la función.
5. El programa continua su ejecución.

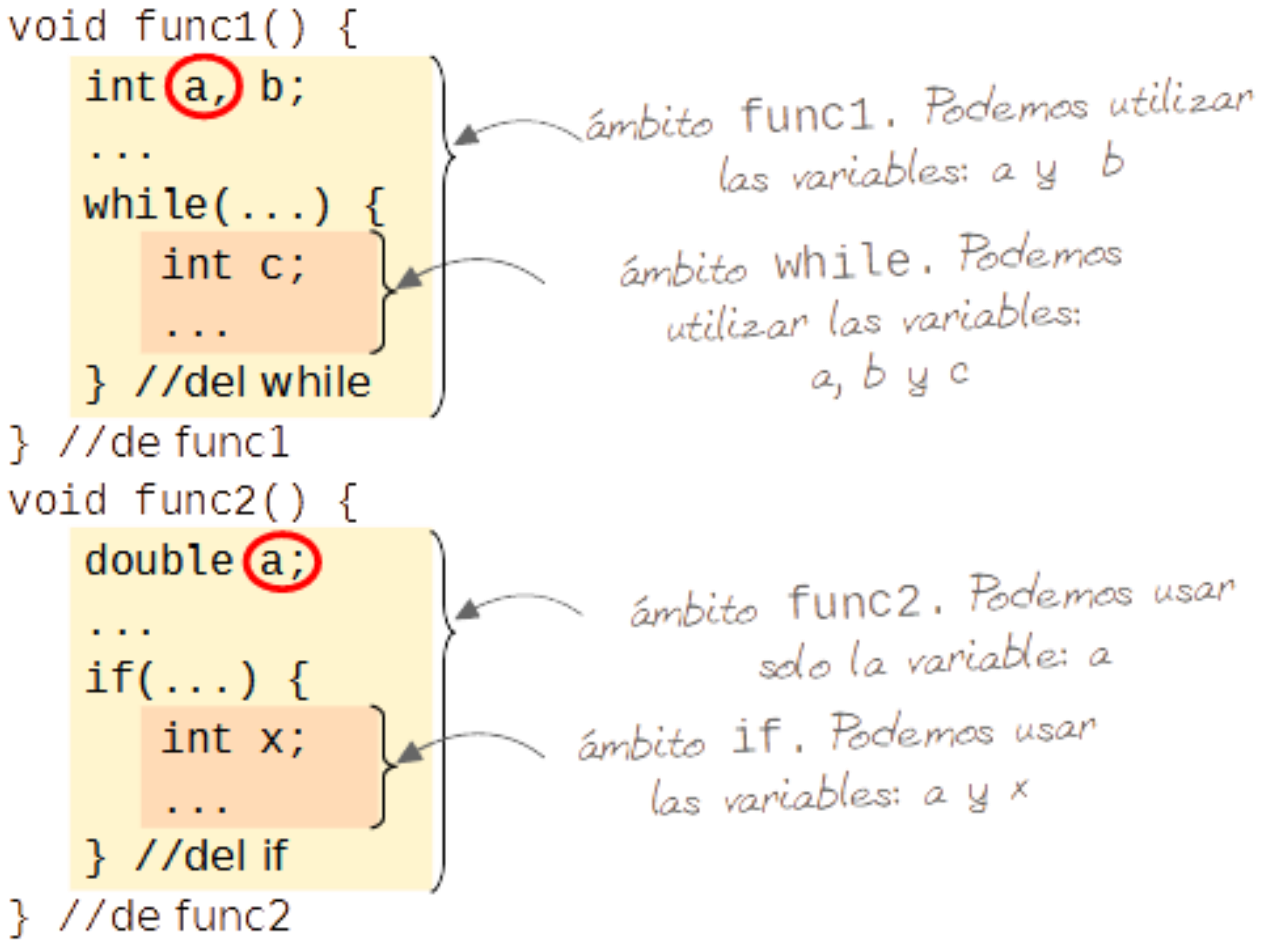
❑ 4.3.2 Métodos – Ámbito de las variables

- En el **cuerpo de una función** podemos declarar **variables**, que se conocen como **variables locales**.

- El **ámbito** de estas, es decir, donde pueden utilizarse, es la propia función donde se declaran, **no pudiéndose utilizar fuera de ella**.

- Nada impide que dentro del cuerpo de una función se utilicen sentencias (**`if`**, **`if-else`**, etc.) con sus respectivos bloques de instrucciones, donde a su vez, se pueden volver a declarar **nuevas variables** que se conocen como **variables de bloques**, siempre y cuando su nombre no coincida con una variable declarada antes, fuera del bloque, ya que esto producirá un **error**.

❑ 4.3.2 Métodos – Ámbito de las variables



❑ 4.3.3 Métodos – Paso de información a un método

- Si deseamos saludar un número distinto de veces, estaríamos obligados a implementar las métodos:
  unSaludo(), **dosSaludos()**, **cuatroSaludos()**, etc.

- Es mucho más práctico implementar la función **variossaludos()** a la que se le pasa el número de veces que deseamos saludar.

```
static void variosSaludos(int veces) {
   for(int i = 0; i < veces; i++) {
      System.out.println("Hola.");
   }
}
```

- De esta manera, si ejecutamos **variossaludos(7)** , saludará siete veces y si ejecutamos **variossaludos(2)** , lo hará en dos ocasiones.

- La variable **veces** es un **parámetro de entrada** de la función **variossaludos()**. Un parámetro de entrada de una función no es más que una variable local a la que se le asigna valores en cada llamada.

❑ 4.3.3 Métodos – Paso de información a un método

```
tipo nombreFuncion(tipo1 parametro1, tipo2 parametro2...) {
   cuerpo de la función
}
```

Sintaxis en java de un **método** con **parámetros de entrada**.

¿De dónde toman los parámetros su valor?

- **Ejemplos**:

```
variosSaludos(2); //llamada con un literal
int n = 3;
variosSaludos(2*n); //llamada con una expresión
```

```
int a = 3;
suma(a, 2); //muestra la suma de a (que vale 3) más 2
```

❑ 4.3.3 Métodos – Paso de información a un método

- **Ejemplo**:

```
static void variosSaludos(int veces) {
   int i;
   //disponemos de las variables locales: i y veces
   //el valor de veces se determina en la llamada
   for(i = 0; i < veces; i++) {
      System.out.println("Hola.");
   }
}
```

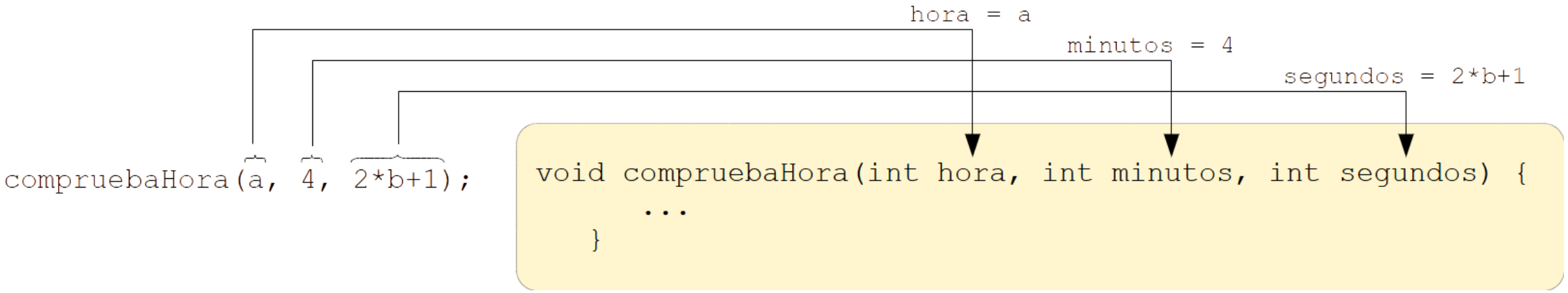
```
variosSaludos(7); //7 se asigna al primer parámetro: veces
```

❑ 4.3.3 Métodos – Paso de información a un método

- **Ejemplo**: (Suponiendo que hubiéramos implementado el método)

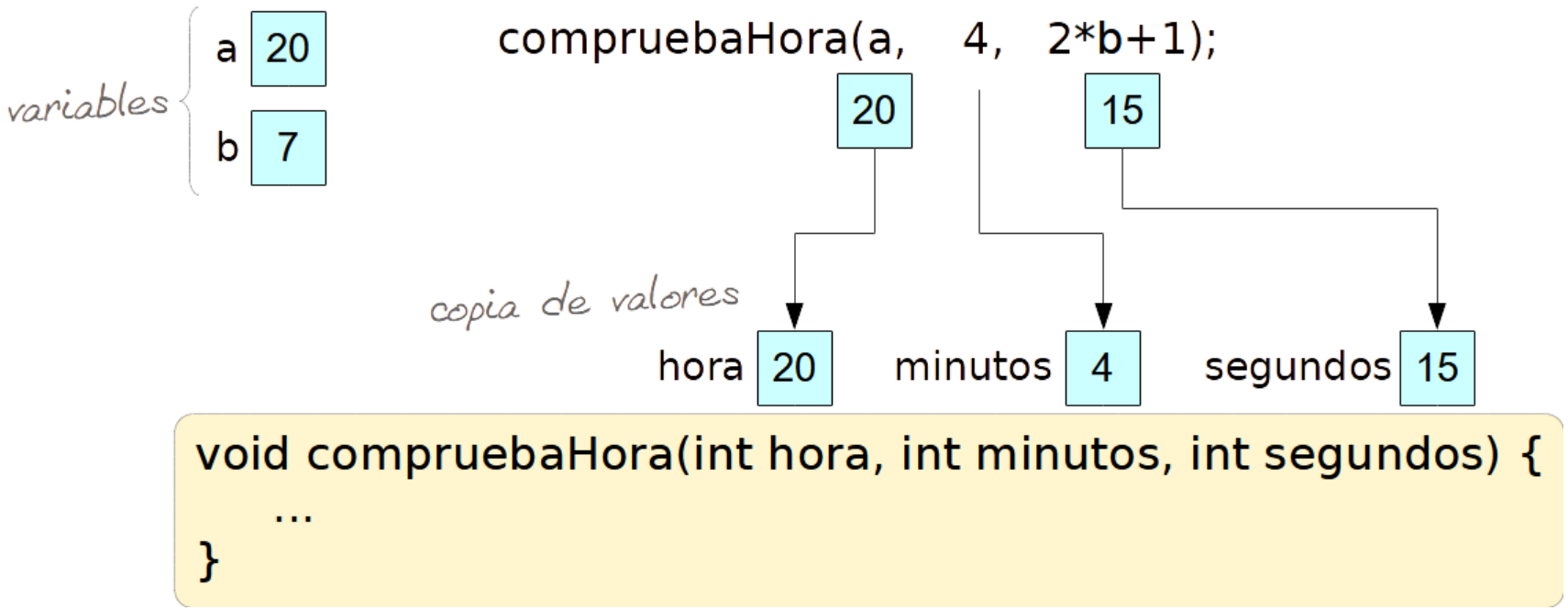
```
static void compruebaHora(int hora, int minutos, int segundos)
```

```
compruebaHora(a, 4, 2*b+1);
```



Paso de parámetros a un método.

❑ 4.3.3 Métodos – Paso de información a un método



Mecanismo de copia de valores a los parámetros

- En **Java** los parámetros toman su valor como una copia del valor de la expresión o variable utilizada en la llamada.
- Este mecanismo de paso de parámetros se denomina **paso de parámetros** por valor o por copia.

4.3.3 Métodos – Paso de información a un método

```
int a = 1, b = 2, c = 3;
compruebaHora(a, b, c); //llamada
...
//definición de la función
static void compruebaHora(int hora, int minutos, int segundos) {
   //hora tiene un valor asignado en la llamada (a=1)
   hora = 23;
   ...
   }
```

¿Qué valor tiene la variable "a" después de modificar la variable "hora"?

❑ 4.3 Métodos

ACTIVIDADES
(void)

1) **Diseñar el método eco() al que se le pase como parámetro un número n, y muestre por pantalla n veces el mensaje "Eco...".**

2) **Escribir un método a la que se le pasen dos enteros y muestre todos los números comprendidos entre ellos.**

3) **Realizar un método que calcule y muestre el área (opción 1) o el volumen (opción 2) de un cilindro, según se especifique. Para distinguir un caso de otro se le pasará como parámetro la opción. Además, hay que pasarle al método el radio y la altura.**

$$\text{área} = 2\pi \cdot \text{radio} \cdot (\text{altura} + \text{radio})$$

$$\text{volumen} = \pi \cdot \text{radio}^2 \cdot \text{altura}$$

❑ 4.3.4 Métodos – Valor devuelto por un método

```
int a = suma(2, 3);
int b = suma(7, 1) * 5;
```

¿Cómo damos valor a la llamada del método?

❑ 4.3.4 Métodos – Valor devuelto por un método

```
tipo nombreFunción(parámetros) {
    ...
    return (valor);
}
```

Sintaxis en java de un **método** con **return**.

- Donde el **`tipo`** de valor debe coincidir con **`tipo`**.
- La instrucción **`return`** se utiliza en **métodos** con un tipo devuelto distinto a **`void`**.

❑ 4.3.4 Métodos – Valor devuelto por un método

- **Ejemplo**: Veamos cómo se implementa la función que realiza la suma de dos números enteros:

```
static int suma(int x, int y) { //cada llamada devuelve un int
    int resultado;
    resultado = x + y;
    return(resultado); //sustituye la llamada por el valor de resultado
}
```

- La última instrucción de la función debe ser **`return`**, que fuerza su fin.
- En caso de existir instrucciones posteriores, no se ejecutarían.
- Nada impide utilizar varios **`return`** en una misma función, pero es una práctica desaconsejable, ya que **un método debe tener un único punto de entrada y de salida**.

❑ 4.3 Métodos

ACTIVIDADES
(return / parámetros)

4) **Diseñar un método que recibe como parámetros dos números enteros y devuelve el máximo de ambos.**

5) **Crear un método que, mediante un booleano, indique si el carácter que se pasa como parámetro de entrada corresponde con una vocal.**

6) **Crea el método `muestraPares(int n)` que muestre por consola los primeros `n` números pares.**

7) **Escribe un método al que se le pase como parámetros de entrada una cantidad de días, horas y minutos. El método calculará y devolverá el número de segundos que existen en los datos de entrada.**

❑ 4.3.5 Métodos – Sobrecarga de métodos

- **Java** permite que dos o más métodos compartan el mismo identificador en un mismo programa. Esto es lo que se conoce como **sobrecarga de métodos**.

- La forma de distinguir entre los distintos métodos sobrecargados es mediante su **lista de parámetros**, que deben ser distintas, ya sean en número o en tipo.

- Los métodos sobrecargados pueden devolver tipos distintos, aunque estos no sirven para distinguir una función sobrecargada de otra.

- Es muy común encontrar en la **API** métodos sobrecargados, ya que permiten agrupar distintas funcionalidades, cuyo uso es similar, bajo el mismo identificador:

    - Por ejemplo, la función que más hemos utilizado hasta ahora, **`System.out.println`**, se encuentra sobrecargada para poder mostrar en pantalla cualquier tipo de dato.

❑ 4.3.5 Métodos – Sobrecarga de métodos

- **Ejemplo**:
  - Queremos diseñar un método para calcular la suma de dos enteros, pero también es útil hacer una suma ponderada, donde cada sumando tenga un peso distinto.
  - Veamos los dos métodos sobrecargados:

```
//función sobrecargada
static int suma(int a, int b) {
   int suma;
   suma = a + b;
   return(suma);
}
```

```
//función sobrecargada
static double suma(int a, double pesoA, int b, double pesoB) {
   double suma;
   suma = a * pesoA / (pesoA + pesoB) + b * pesoB / (pesoA + pesoB);
   return(suma);
}
```

❑ 4.3.5 Métodos – Sobrecarga de métodos

- A partir de la definición de los métodos, ambas están disponibles, y cumplen con la única restricción de los métodos sobrecargados: que **se puedan distinguir mediante sus parámetros**:
  - Si invocamos el método de la forma: **`suma(2, 3)`**

```
//función sobrecargada
static int suma(int a, int b) {
   int suma;
   suma = a + b;
   return(suma);
}
```

¿Qué devolverán?

  - Si se llama con: **`suma(2, 0.25, 3, 0.75)`**

```
//función sobrecargada
static double suma(int a, double pesoA, int b, double pesoB) {
   double suma;
   suma = a * pesoA / (pesoA + pesoB) + b * pesoB / (pesoA + pesoB);
   return(suma);
}
```

❑ 4.3.5 Métodos – Sobrecarga de métodos

ACTIVIDADES

8) **Repetir la Actividad 4 (***Diseñar un método que recibe como parámetros dos números enteros y devuelve el máximo de ambos***) con una versión que calcule el máximo de tres números.**

❑ 4.3.6 Métodos – Recursividad

- Un **método** puede ser invocada desde cualquier lugar:
  - desde el programa principal.
  - desde otro método.
  - incluso desde dentro de su propio cuerpo de instrucciones.

- En este último caso, cuando un método se invoca a sí mismo, diremos que es un **método o función recursiva**.

```
static int funcionRecursiva() {
    ...
    funcionRecursiva(); //llamada recursiva
    ...
}
```

*Sintaxis en java de un **método recursivo**.*

¿Cuándo para?

4.3.6 Métodos – Recursividad

- Para evitarlo, hemos de habilitar un mecanismo que detenga, en algún momento, la serie de llamadas recursivas: una sentencia **`if`** que, utilizando una condición, llamada «**caso base**», impida que se continúe con una nueva llamada recursiva. Veamos el esquema general:

```
int funcionRecursiva(datos) {
   int resultado;
   if (caso base) {
      resultado = valorBase;
   } else {
      resultado = funcionRecursiva(nuevosDatos); //llamada recursiva
      ...
   }
   return (resultado);
}
```

- Solo cuando la condición del caso base sea **`false`**, se hará una nueva llamada recursiva.
- Cuando el caso base sea **`true`** se romperá la cadena de llamadas. La idea principal de

❑ 4.3.6 Métodos – Recursividad

- **Ejemplo #1**:
  - Imagina que cada paso de una **escalera** representa un "nivel" y queremos saber cuántos pasos quedan por dar para llegar abajo. La recursividad aquí permite ver cómo cada paso se cuenta hasta alcanzar el final de la escalera.

```java
public class BajarEscalera {
    public static void main(String[] args) {
        int escalones = 10; // Número total de escalones
        bajarEscalera(escalones);
    }

    // Método recursivo para bajar un escalón cada vez
    public static void bajarEscalera(int escalones) {
        if (escalones == 0) { // Condición de parada: ya estamos abajo
            System.out.println("¡Has llegado al final de la escalera!");
        } else {
            System.out.println("Quedan " + escalones + " escalones.");
            bajarEscalera(escalones - 1); // Llamada recursiva restando un escalón
        }
    }
}
```

❑ 4.3.6 Métodos – Recursividad

- **Ejemplo #2:**
  - Supongamos que deseamos calcular el **factorial** de un número **n**, que se representan por **n!**
  - Sabemos que:

$$n! = n * (n - 1) * (n - 2) * \dots 2 * 1$$

  - Un ejemplo para calcular 5 factorial sería:

$$5! = 5 * 4 * 3 * 2 * 1$$

  - Para calcular el **factorial** de un número, estamos utilizando el **factorial** de un número más pequeño, con lo cual estamos reduciendo el problema.
  - Hemos de buscar un **caso base**, es decir, un valor para el que calcular el factorial sea algo trivial y no necesitemos volver a utilizar el método recursivo.
  - Se considera por definición que el **factorial** de **0** vale **1**.
  - El **caso base** del factorial es: $0! = 1$.

4.3.6 Métodos – Recursividad

- **Ejemplo #2:**

```
long factorial(int n) {
    long resultado;
    if (n == 0) { //si n es 0
      resultado = 1; //caso base
    } else {
      resultado = n * factorial(n - 1); //llamada recursiva
    }
    return(resultado);
}
```

❑ 4.3.6 Métodos – Recursividad

- **Ejemplo #2:**

```
long factorial(3) {
   long resultado;
   if (3 == 0) { //falso
      ...
   } else {
      resultado = 3 * factorial(2);
```

```
long factorial(2) {
   long resultado;
   if (2 == 0) { //falso
      ...
   } else {
      resultado = 2 * factorial(1);
```

```
long factorial(1) {
   long resultado;
   if (1 == 0) { //falso
      ...
   } else {
      resultado = 1 * factorial(0);
```

```
long factorial(0) {
   long resultado;
   if (0 == 0) { //cierto
      resultado = 1;
   } else {
      ...
   }
   return(1);
}
```

❑ 4.3.6 Métodos – Recursividad

ACTIVIDADES

9) **Diseñar una función que calcule a^n, donde a es real y n es entero no negativo. Realizar una versión iterativa y otra recursiva.**

10) **Diseñar una función recursiva que calcule el enésimo término de la serie de Fibonacci. En esta serie el enésimo valor se calcula sumando los 2 valores anteriores de la serie.**

Es decir:

fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
fibonacci(0) = 1
fibonacci(1) = 1