Capítulo 1
Estructuras básicas de programación

El propósito general de este capítulo es conocer las estructuras básicas de programación que tiene el lenguaje de programación Java para especificar el flujo de control de los programas. Estas estructuras básicas forman el núcleo del lenguaje y serán ampliamente utilizadas en el resto de los ejercicios del libro.

En este primer capítulo se presentan cinco ejercicios sobre estructuras básicas de programación utilizando Java.

- **Ejercicio 1.1. Estructura condicional *if-else***
  La estructura condicional *if-else* permite ejecutar diferentes porciones de código basadas en una condición booleana. El primer ejercicio trata sobre la utilización de *if* anidados.

- **Ejercicio 1.2. Estructura repetitiva *while***
  En el segundo ejercicio se abordan las estructuras repetitivas, iniciando con la estructura *while* que permite repetir una instrucción o bloque de instrucciones siempre que una condición particular sea verdadera. El segundo ejercicio coloca en práctica la aplicación de la estructura *while*.

- **Ejercicio 1.3. Estructura repetitiva *do-while***
  Otra estructura repetitiva es el *do-while*, que repite una instrucción o bloque de instrucciones siempre y cuando se cumpla cierta condición booleana. Se diferencia del

while en que la instrucción o bloque se ejecuta al menos una vez. El tercer ejercicio aborda la utilización de la estructura *do-while*.

- **Ejercicio 1.4. Estructura repetitiva *for***
  En la misma línea, otra estructura repetitiva muy utilizada es el *for*, el cual repite una instrucción o un bloque de instrucciones varias veces hasta que una condición se cumpla. El cuarto ejercicio propone un algoritmo que utiliza dicha estructura.

- **Ejercicio 1.5. *Arrays***
  Por último, el concepto de *array* es extensamente aplicado en numerosas soluciones de algoritmos para agrupar una colección de elementos de un mismo tipo. Por ello, se presenta un ejercicio que aplica este concepto.

Los diagramas UML utilizados en este capítulo son los diagramas de actividad. Las soluciones presentadas están más orientadas a la programación estructurada que a la programación orientada a objetos. Por ello, se utilizan los diagramas de actividad UML, para modelar el flujo de control de los algoritmos y podrían reemplazar a los típicos diagramas de flujo.

Ejercicio 1.1. Estructura condicional *if-else*

La declaración *if* es la más básica de todas las declaraciones de flujo de control en Java. Esta sentencia le indica al programa que ejecute una determinada sección de código solo si una condición en particular se evalúa como verdadera (Vozmediano, 2017). El formato de la instrucción es el siguiente:

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

Si la condición se evalúa como falsa, el control salta al final de la declaración *if*.

Otro formato de esta instrucción es utilizando *if-else*, el cual proporciona una ruta secundaria cuando la condición se evalúa como falsa. El formato es el siguiente:

```
if (condición) {
```

```
        bloque de instrucciones
} else {
        bloque de instrucciones
}
```

Además, la estructura *if-else* se puede anidar, es decir, que se puede usar una declaración *if* o *else-if* dentro de otra declaración *if* o *else-if*. El formato es el siguiente:

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

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Entender el concepto de flujo de control condicional.
- Definir estructuras condicionales utilizando Java.
- Definir estructuras condicionales anidadas.

Enunciado: índice de masa corporal

Se desea desarrollar un programa que calcule el índice de masa corporal de una persona. Para ello, se requiere definir el peso de la persona (en kilogramos) y su estatura (en metros). El índice de masa corporal (IMC) se calcula utilizando la siguiente fórmula:

$$IMC = \frac{peso}{estatura^2}$$

Luego, a partir del IMC obtenido se pueden calcular si una persona tiene un peso normal, inferior o superior al normal u obesidad. Para generar estos resultados el IMC calculado debe estar en los rangos de la tabla 1.1.

Tabla 1.1. Cálculo de IMC

IMC	Resultado	IMC	Resultado
< 16	Delgadez severa	[25-30)	Sobrepeso
[16-17)	Delgadez moderada	[30-35)	Obesidad leve
[17-18.5)	Delgadez leve	[35-40)	Obesidad moderada
[18.5-25)	Peso normal	>=40	Obesidad mórbida

Instrucciones Java del ejercicio

Tabla 1.2. Instrucciones Java del ejercicio 1.1.

Instrucción	Descripción	Formato
main	Método que define el punto de inicio de un programa. Es un método que no retorna ningún valor.	*public static void main(String args[])*
System.out.println	Método que imprime en pantalla el argumento que se le pasa como parámetro y luego realiza un salto de línea.	*System.out.println*(argumento);
Math.pow	Método que devuelve el valor del primer argumento elevado a la potencia del segundo argumento.	*Math.pow*(argumento1, argumento2);

Solución

Clase: IndiceIMC

```
/**
* Esta clase denominada IndiceIMC calcula el índice de masa corporal de
* una persona y con base en el resultado indica si tiene un peso normal,
* inferior o superior al normal u obesidad.
* @version 1.0/2020
*/
public class IndiceIMC {
```

```
    /**
     * Método main
     */
    public static void main(String args[]) {
        int masa = 91; // Masa en kilogramos
        double estatura = 1.77; // Estatura en metros
        double IMC = masa/Math.pow(estatura, 2); /* Calcular el índice
            de masa corporal */
        System.out.println("La persona tiene una masa = " + masa + "
            kilogramos y estatura = " + estatura + " metros"); /* Mediante
            varios if-else anidados se evalúan diferentes rangos del IMC */
        if (IMC < 16) {
            System.out.println("La persona tiene delgadez severa.");
        } else if (IMC < 17) {
            System.out.println("La persona tiene delgadez moderada.");
        } else if (IMC < 18.5) {
            System.out.println("La persona tiene delgadez leve.");
        } else if (IMC < 25) {
            System.out.println("La persona tiene peso normal.");
        } else if (IMC < 30) {
            System.out.println("La persona tiene sobrepeso.");
        } else if (IMC < 35) {
            System.out.println("La persona tiene obesidad leve.");
        } else if (IMC < 40) {
            System.out.println("La persona tiene obesidad media.");
        } else {
            System.out.println("La persona tiene obesidad mórbida.");
        }
    }
}
```

Diagrama de actividad

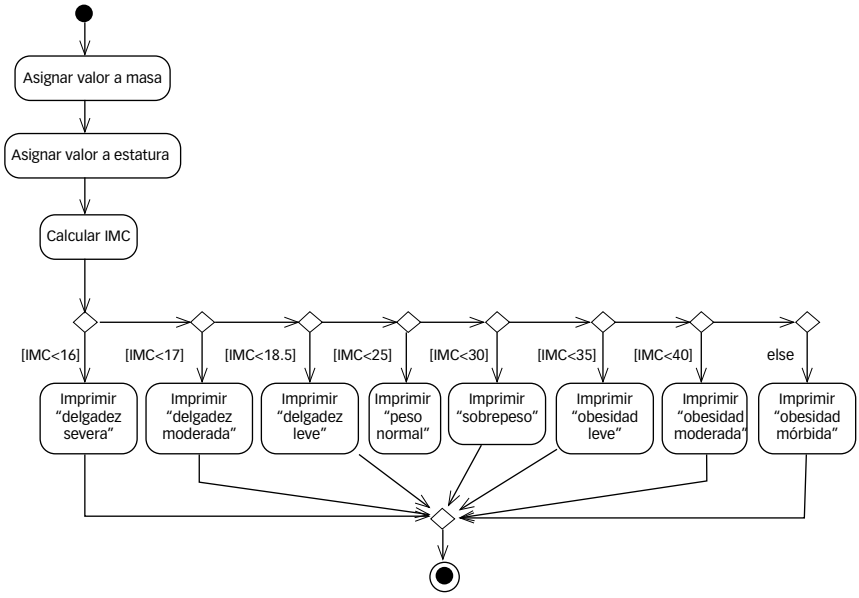

Figura 1.1. Diagrama de actividad del ejercicio 1.1.

Explicación del diagrama de actividad

El diagrama de actividad UML muestra el algoritmo del programa desarrollado para calcular el índice de masa corporal y mostrar un mensaje al usuario de acuerdo con el IMC obtenido.

El diagrama posee una actividad inicial representada por medio de un círculo negro al inicio del programa.

Las instrucciones del programa son "actividades" que se representan en UML con un rectángulo con los bordes redondeados. Mediante flechas se establece el flujo de control del programa y por medio de rombos se definen comportamientos condicionales que, si se cumplen, se bifurcarán por la ruta indicada en el flujo de control del programa.

El programa inicia con la asignación de valores a las variables masa y estatura. Luego, se calcula el IMC. Después, de acuerdo con el IMC calculado, se establece una serie de *ifs* (representados por medio de rombos que muestran bifurcaciones con expresiones booleanas encerradas entre []), si una de dichas condiciones se cumple, se imprime un texto indicando el estado de la persona.

Por lo tanto, el programa está conformado por una serie de *ifs* anidados, donde si un *if* no se cumple, se pasa a evaluar el siguiente y así sucesivamente, hasta finalizar el programa, el cual se representa como una actividad final con círculos concéntricos negro y blanco.

Ejecución del programa

```
La persona tiene una masa = 91 kilogramos y estatura = 1.77 metros
La persona tiene sobrepeso.
```

Figura 1.2. Ejecución del programa del ejercicio 1.1.

Ejercicios propuestos

- Hacer un programa que calcule las raíces de una ecuación cuadrática.
- Hacer un programa que, dado el número de un mes, presente el nombre del mes y determine la cantidad de días que tiene.
- Hacer un programa que determine si un año es bisiesto o no.

Ejercicio 1.2. Estructura repetitiva *while*

La sentencia *while* ejecuta continuamente un bloque de instrucciones mientras una condición es verdadera (Altadill-Izurra y Pérez-Martínez, 2017). El formato de la expresión es:

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

La instrucción *while* evalúa la condición que debe devolver un valor booleano. Si la expresión se evalúa como verdadera, la instrucción *while* ejecuta el bloque de instrucciones. La instrucción *while* continúa evaluando la expresión y ejecutando el bloque de instrucciones hasta que la condición se evalúe como falsa.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Comprender el significado y aplicación de la sentencia *while*.
- Desarrollar algoritmos que implementen dicha sentencia.

Enunciado: número de Amstrong

Se quiere desarrollar un programa que determine si un número es un número de Amstrong. Un número de Amstrong es aquel que es igual a la suma de sus dígitos elevados a la potencia de su número de cifras.

Por ejemplo, el número 371 es un número que cumple dicha característica ya que tiene tres cifras y:

$$371 = 3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371$$

Instrucciones Java del ejercicio

Tabla 1.3. Instrucciones Java del ejercicio 1.2.

Instrucción	Descripción	Formato
Math.floor	Devuelve el máximo entero menor o igual a un número pasado como parámetro.	*Math.pow*(x);
Math.log10	Devuelve el logaritmo en base 10 de un número pasado como parámetro.	*Math.log10*(x);
%	Operador resto de la división entera.	número % número
Math.pow	Método que devuelve el valor del primer argumento elevado a la potencia del segundo argumento.	*Math.pow*(argumento1, argumento2);

Solución

Clase: NúmeroAmstrong

```
/**
* Esta clase denominada NúmeroAmstrong determina si un número
* entero cumple el requerimiento que la suma de sus dígitos elevado a su
* cantidad de dígitos es el mismo número.
* @version 1.0/2020
*/
```

```
public class NúmeroAmstrong {

    /**
    * Método main
    */
    public static void main(String args[]) {
        int númeroOriginal, últimoDigito; /* Variables para el número
            original y su último dígito */
        double dígitos; // Cantidad de dígitos que tiene el número
        double suma = 0; /* Variable que sumará los dígitos elevados a su
            cantidad de dígitos */
        int número = 371; /* Número a determinar si es un número de
            Amstrong */

        númeroOriginal = número; /* Copia el valor del número para su
            procesamiento */
        dígitos = Math.floor(Math.log10(número)) + 1; /* Calcula el total
            de dígitos del número */
        // Calcula la suma de potencia de digitos
        while (número > 0) {
            últimoDigito = número % 10; // Extrae el último dígito
            // Calcula la suma de potencias del último dígito
            suma = suma + Math.pow(últimoDigito, dígitos);
            número = número / 10; // Elimina el último dígito
        }

        */ Verifica si es un número de Armstrong si la suma obtenida es
            igual al número */
        if (númeroOriginal == suma) {
            System.out.println(númeroOriginal + " es un número de
                Amstrong");
        } else {
            System.out.println(númeroOriginal + " no es un número de
                Amstrong");
        }
    }
}
```

Diagrama de actividad

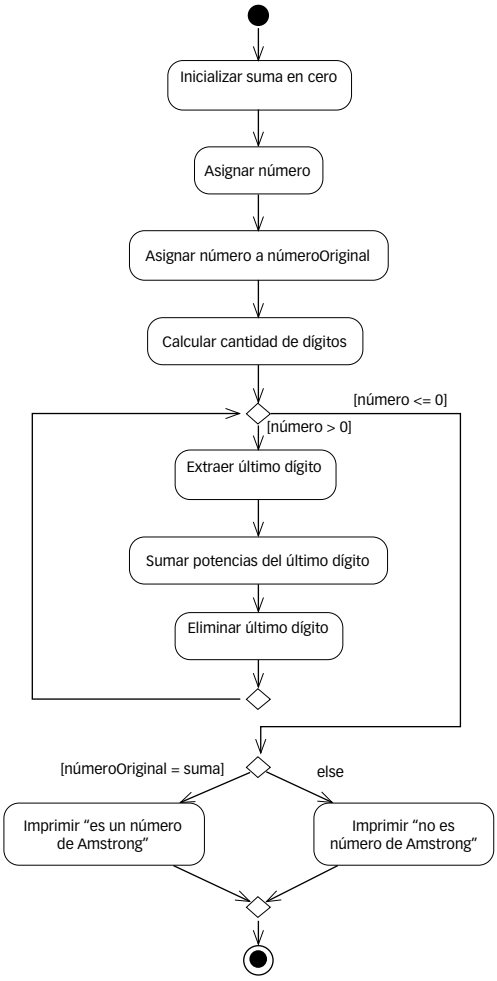

Figura 1.3. Diagrama de actividad del ejercicio 1.2.

Explicación del diagrama de actividad

La figura 1.3 muestra, como un diagrama de actividad UML, el algoritmo de la solución del ejercicio para determinar si un número dado es un número de Amstrong.

El diagrama posee una actividad inicial representada por medio de un círculo negro al inicio del programa.

Las diferentes instrucciones del programa se modelan como "actividades", las cuales se representan en UML como rectángulos con los bordes redondeados. El programa identifica como actividades las instrucciones de inicializar variables, asignar valores a las variables y calcular la cantidad de dígitos.

El ciclo *while* está representado por medio de una pareja de rombos que marcan el inicio y fin del ciclo. Por lo tanto, mientras el número sea mayor que cero, se ejecutarán en forma iterativa las actividades: extraer último dígito, sumar potencias del último dígito y eliminar último dígito. Cuando el número sea menor que cero, finaliza el ciclo y se evalúa en un *if* si la suma obtenida es el número original. Si se cumple está condición, se imprime que es un número de Amstrong. En caso contrario, se imprime el mensaje correspondiente. Con estas impresiones finaliza el programa, el cual se representa por medio de un círculo blanco y negro concéntrico.

Ejecución del programa

```
371 es un número de Amstrong
```

Figura 1.4. Ejecución del programa del ejercicio 1.2.

Ejercicios propuestos

- Escribir un programa que dado un número entero positivo *n*, calcule la suma de la siguiente serie: 1 + 1/2 + 1/3 + 1/4 + 1/5 + … + 1/n
- Escribir un programa que calcule los primeros *n* números de Fibonacci. Los números de Fibonacci comienzan con 0 y 1, y cada término siguiente es la suma de los anteriores: 0, 1, 2, 3, 5, 8, 13, 21, …

Ejercicio 1.3. Estructura repetitiva *do-while*

La sentencia *do-while* también es una estructura repetitiva que se ejecuta siempre y cuando se cumpla cierta condición booleana. La diferencia entre *do-while* y *while* es que la estructura *do-while* evalúa su expresión en la parte inferior del bucle en lugar de la parte superior (Bloch, 2017). Por lo tanto, si se utiliza la estructura *do-while*, el bloque de instrucciones se ejecutará por lo menos una vez. El formato de la instrucción es la siguiente:

do {
bloque de instrucciones
} while (condición)

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Entender el concepto de la sentencia *do-while*.
- Definir estructuras repetitivas utilizando la sentencia *do-while*.
- Diferenciar la estructura repetitiva *while* y *do-while*.

Enunciado: número perfecto

Se quiere desarrollar un programa que determine si un número es un número perfecto. Un número perfecto es aquel que es igual a la suma de sus divisores positivos.

Por ejemplo, el número 28 es un número perfecto ya que sus divisores son: 1, 2, 4, 7 y 14, y la suma de estos números es 28.

Instrucciones Java del ejercicio

Tabla 1.4. Instrucciones Java del ejercicio 1.3.

Instrucción	Descripción	Formato
%	Operador resto de la división entera.	número % número
++	Operador de incremento, utilizado para incrementar un valor en 1. Tiene dos variedades: preincremento (el valor se incrementa primero y luego se calcula el resultado) y posincremento (el valor se usa por primera vez para calcular el resultado y luego se incrementa).	Preincremento: variable++; Posincremento: ++variable;

Solución

Clase: NúmeroPerfecto

```
/**
 * Esta clase denominada NúmeroPerfecto determina si la suma de los
 * divisores de un número es el mismo número.
 * @version 1.0/2020
 */
```

```
public class NúmeroPerfecto {

    /**
    * Método main
    */
    public static void main(String args[]) {
        int suma = 0; // Variable que sumará los divisores del número
        int número = 496; // Número a determinar si es perfecto o no
        int i = 1; /* Variable utilizada para determinar los divisores del
            número */
        // Calcula la suma de todos los divisores
        do {
            // Si i es un divisor del número, se va acumulando
            if (número % i == 0) {
                suma = suma + i;
            }
            i++;
        } while (i <= número / 2); /* No existen divisores mayores a la
            mitad del número */
        // Verifica si la suma de los divisores del número es igual al número
        if (suma == número) {
            System.out.println(número + " es un número perfecto");
        } else {
            System.out.println(número + " no es un número perfecto");
        }
    }
}
```

Diagrama de actividad

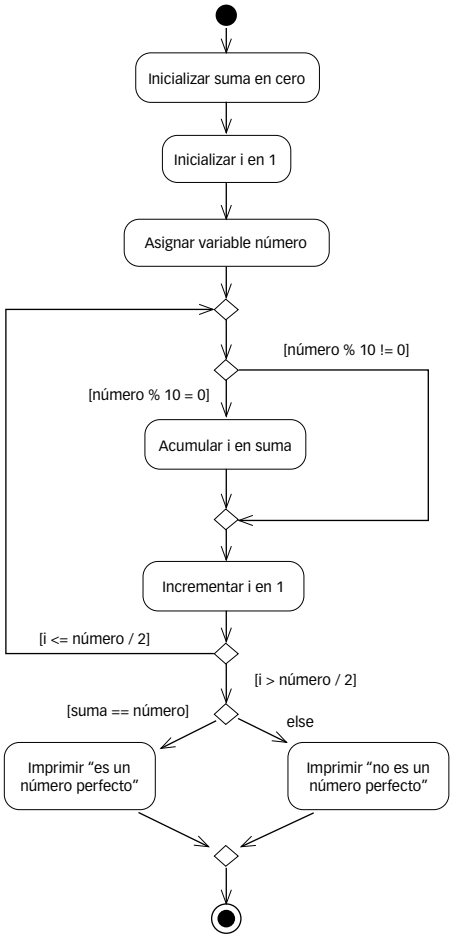

Figura 1.5. Diagrama de actividad del ejercicio 1.3.

Explicación del diagrama de actividad

El diagrama de actividad UML presentado muestra un modelo del flujo de control del algoritmo para determinar si un número es perfecto o no.

El diagrama posee una actividad inicial representada por medio de un círculo negro al inicio del programa.

Las diferentes instrucciones del programa se modelan como "actividades", las cuales se representan en UML como rectángulos con los bordes

redondeados. El programa identifica como actividades las instrucciones de inicializar variables y asignar valores a las variables.

El ciclo *do-while* está representado por medio de una pareja de rombos que marcan el inicio y fin del ciclo. Por lo tanto, mientras la condición (variable i < número /2) se cumpla, se ejecutarán en forma iterativa la actividad condicional de acumular i en la variable suma, la cual se ejecuta si el residuo entre el número y 10 es cero. Se puede observar que este ciclo se ejecutará por lo menos una vez en el programa.

Se evalúa si la suma obtenida es igual al número al terminar el ciclo *do-while*, si se cumple esta condición se imprime o no el mensaje indicando si el número es perfecto o no. Con estas impresiones finaliza el programa, el cual se representa por medio de un círculo blanco y negro concéntrico.

Ejecución del programa

```
496 es un número perfecto
```

Figura 1.6. Ejecución del programa del ejercicio 1.3.

Ejercicios propuestos

- Escribir un programa que, dado un número, determine cuántos dígitos tiene.
- Escribir un programa que, dadas 5 notas finales, determine cuántas notas fueron mayores o iguales a 3.0.

Ejercicio 1.4. Estructura repetitiva *for*

La instrucción *for* proporciona una forma compacta de iterar sobre un rango de valores. Este tipo de ciclo se repite hasta que se cumple una condición particular (Cadenhead, 2017). La forma general de la declaración *for* es:

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

La expresión de inicialización da inicio al ciclo y se ejecuta una vez, cuando comienza el ciclo. El ciclo termina cuando la expresión de terminación se

evalúa como falsa. La expresión de incremento se invoca después de cada iteración a través del ciclo e incrementa o disminuye un determinado valor.

El ciclo *for* se utiliza con bastante frecuencia en iteraciones simples en las que se repite un bloque de instrucciones un cierto número de veces y luego se detiene, pero se pueden utilizar ciclos *for* para casi cualquier tipo de ciclo.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Comprender el significado y aplicación de la sentencia *for*.
- Desarrollar algoritmos que implementen dicha sentencia.

Enunciado: números amigos

Se quiere desarrollar un programa que determine si dos números son amigos. Dos números enteros positivos se consideran amigos si la suma de los divisores de uno es igual al otro número y viceversa.

Por ejemplo, los números 220 y 284 son amigos. Los divisores del número 220 son: 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 y 110, y suman 284. Los divisores de 284 son: 1, 2, 4, 71 y 142, que suman 220.

Instrucciones Java del ejercicio

Tabla 1.5. Instrucciones Java del ejercicio 1.4.

Instrucción	Descripción	Formato
%	Operador resto de la división entera.	número % número

Solución

Clase: NúmerosAmigos

```
/**
 * Esta clase denominada Números amigos determina si una pareja de
 * números son amigos. Dos números enteros positivos son amigos si la
 * suma de los divisores propios de un número es igual al otro número
 * y viceversa.
 * @version 1.0/2020
 */
```

```
public class NúmerosAmigos {

    /**
    * Método main
    */
    public static void main(String[] args) {
        int suma = 0; // Variable que sumará los divisiores de un número
        int número1 = 220; // Definición del primer número
        int número2 = 284; // Definición del segundo número
        // Suma todos los divisores del número 1
        for(int i = 1; i < número1; i++) {
            if (número1 % i == 0) {
                suma = suma + i;
            }
        }
        // Si la suma de los divisores del número 1 es igual al número 2
        if (suma == número2) {
            suma = 0;
            // Suma los divisores del número 2
            for(int i = 1; i < número2; i++) {
                if (número2 % i == 0) {
                suma= suma + i;
            }
        }
        // Si la suma de los divisores de ambos números son iguales
        if (suma == número1) {
            System.out.println(número1 + " y " + número2 + " son ami-
                gos");
        } else {
            System.out.println(número1 + " y " + número2 + " no son
                amigos");
            }
        } else {
            System.out.println(número1 + " y " + número2 + " no son
                amigos");
        }
    }
}
```

Diagrama de actividad

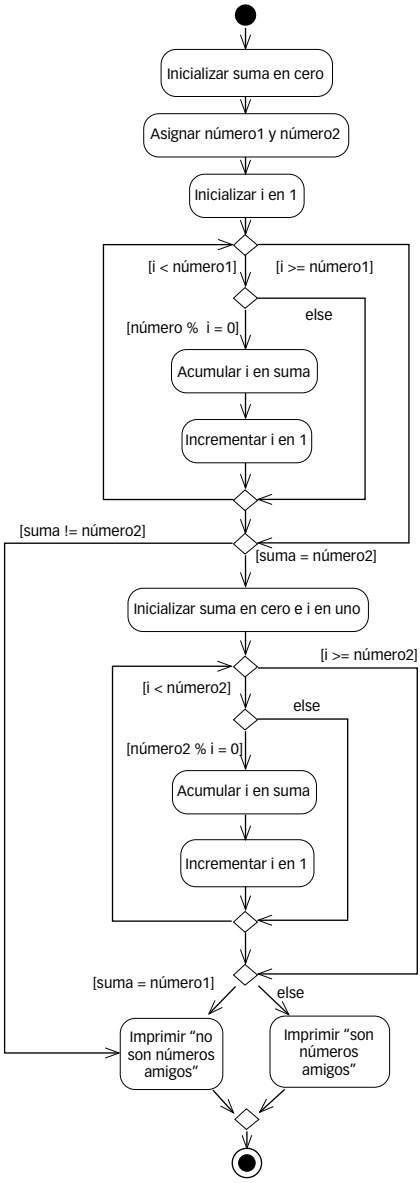

Figura 1.7. Diagrama de actividad del ejercicio 1.4.

Explicación del diagrama de actividad

El diagrama de actividad UML presentado muestra un modelo del flujo de control del algoritmo para determinar si un par de números se consideran amigos o no.

El diagrama posee una actividad inicial representada por medio de un círculo negro al inicio del programa.

Las diferentes instrucciones del programa se modelan como "actividades", las cuales se representan en UML como rectángulos con los bordes redondeados. El programa identifica como actividades las instrucciones de inicializar la variable suma, asignar valores a las variables número1 y número2 e inicializar i en 1.

Los ciclos *for* se representan por medio de una pareja de rombos que marcan el inicio y fin del ciclo. Por lo tanto, la actividad condicional (si el residuo entre número1 e i es cero) se ejecutará en un ciclo desde i = 1 hasta que sea igual a número1. Si la condición se cumple, se acumula el valor de i en la variable suma.

Al final este ciclo *for*, se evalúa si la suma obtenida es igual al número 2. Si dicha condición se cumple, se ingresa a un segundo ciclo, que se repite desde i igual a uno hasta que i sea igual a número2. Durante dicho ciclo se acumula el valor de i en suma. Al finalizar el ciclo, se evalúa si la suma obtenida es igual al número2. De acuerdo con esta condición, se imprime o no el mensaje indicando si los números son amigos o no. Con estas impresiones finaliza el programa, el cual se representa por medio de un círculo blanco y negro concéntrico.

Ejecución del programa

```
220 y 284 son amigos
```

Figura 1.8. Ejecución del programa del ejercicio 1.4.

Ejercicios propuestos

- Desarrollar un programa que calcule el factorial de un número entero positivo. El factorial de un número es el producto de todos los números enteros positivos desde 1 hasta el número en cuestión.

- Desarrollar un programa de determine el máximo común divisor y el mínimo común múltiplo de un número.

Ejercicio 1.5. *Arrays*

Un *array* es un objeto que contiene un número fijo de valores de un solo tipo. La longitud de un *array* se establece cuando se crea el *array*. Después de su creación, su longitud es fija y no puede cambiar (Clark, 2017).

Cada elemento de un *array* se accede por medio de su índice numérico. Los índices numéricos comienzan con 0. Por ejemplo, el séptimo elemento se accedería a través del índice 6 como se observa en la figura 1.9.

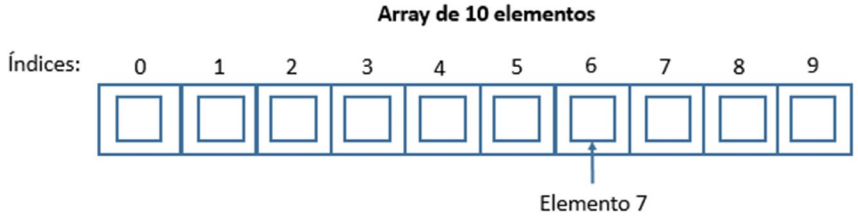

Figura 1.9. Estructura de un *array* con sus índices y elementos

Una *array* se declara de la siguiente forma:

tipoDato [] nombreArray;

El *array* se crea utilizando el operador *new:*

nombreArray = new int[tamaño];

donde tamaño es un valor entero que representa el tamaño máximo del *array*.

Los siguientes formatos de instrucciones asignan valores a cada elemento del *array*:

```
nombreArray[0] = valor; // Inicializa el primer elemento
nombreArray[1] = valor; // Inicializa el segundo elemento
nombreArray[2] = valor; // Inicializa el tercer elemento
```

Una forma alternativa y abreviada de inicializar un *array* es proporcionar sus elementos entre llaves y separados por comas. Por ejemplo, un *array* de valores enteros:

```
int[] unArray = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
```

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Entender y aplicar el concepto de *array*.
- Desarrollar algoritmos que implementen en su solución el concepto de *array*.

Enunciado: elementos duplicados

Se desea desarrollar un programa que, dado un *array* de números enteros, determine cuáles son sus elementos que se encuentran duplicados.

Instrucciones Java del ejercicio

Tabla 1.6. Instrucciones Java del ejercicio 1.5.

Instrucción	Descripción	Formato
length	Método que determina la longitud o tamaño de un *array*.	nombreArray.length()
&&	Operador AND lógico, devuelve verdadero cuando ambas condiciones son verdaderas; de otra manera, falso.	expresión1 && expresión2
!=	Operador diferente o no igual a. Devuelve verdadero si el valor del lado izquierdo no es igual al lado derecho.	variable1 != variable2

Solución

Clase: ElementosDuplicados

```
/**
 * Esta clase denominada ElementosDuplicados permite detectar cuáles
 * son los elementos duplicados en un array.
 * @version 1.0/2020
 */
```

```
public class ElementosDuplicados {

    /**
    * Método main
    */
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 3, 4, 4, 5, 2}; /* Definición de un array de
            datos int */
        System.out.println("Elementos del array");
        // Imprime los elementos del array
        for (int i = 0; i < array.length; i++) {
            System.out.println("Elemento [" + i + "] = " + array[i]);
        }
        for (int i = 0; i < array.length - 1; i++) { /* Primer ciclo que recorre
            el array */
            for (int j = i+1; j < array.length; j++) { /* Segundo ciclo que
                recorre el array */
                /* Evalúa si los elementos son iguales y están en posiciones
                    diferentes */
                if ((array[i] == array[j]) && (i != j)) {
                System.out.println("Elemento duplicado: " + array[j]);
                }
            }
        }
    }
}
```

Diagrama de actividad

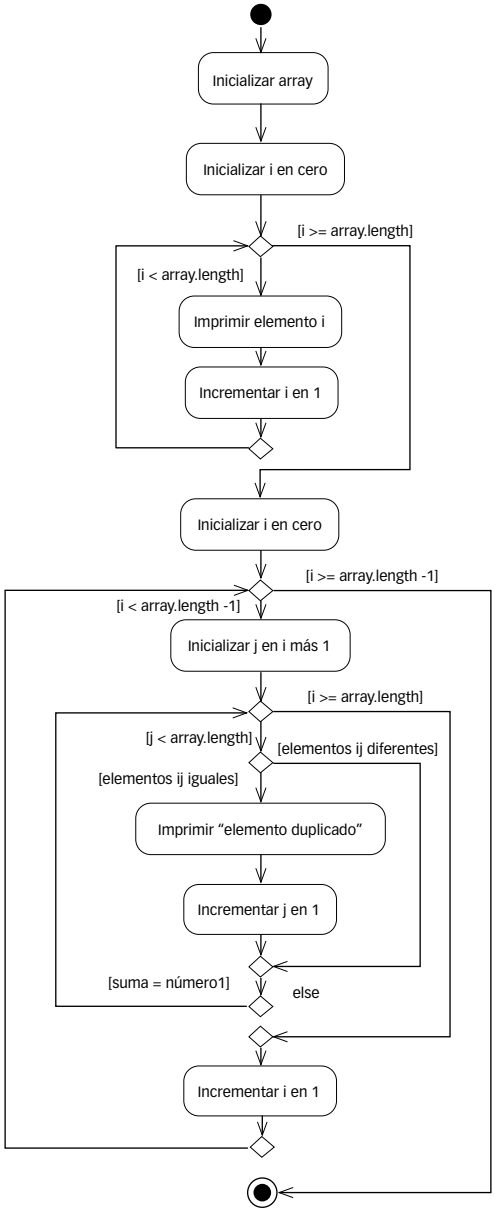

Figura 1.10. Diagrama de actividad del ejercicio 1.5.

Explicación del diagrama de actividad

El diagrama de actividad UML muestra un modelo del flujo de control del algoritmo para determinar los elementos duplicados de un *array* de valores enteros.

El diagrama posee una actividad inicial representada por medio de un círculo negro al inicio del programa.

Las diferentes instrucciones del programa se modelan como "actividades", las cuales se representan en UML como rectángulos con los bordes redondeados. El programa identifica como actividades las instrucciones de inicializar el *array* e inicializar la variable i.

El ciclo *for* que imprime los valores del *array* se representan por medio de una pareja de rombos que marcan el inicio y fin del ciclo.

Luego se presentan dos ciclos *for* anidados. El primero recorre el *array* desde la posición i = 0 hasta la longitud del *array* -1, y el segundo recorre el *array* desde j = i+1 hasta la longitud del *array*. En estos ciclos se evalúa que, si los elementos son iguales, se imprima el mensaje que el elemento está duplicado.

Cuando finalizan estos dos ciclos *for* anidados termina el programa, el cual se representa por medio de un círculo blanco y negro concéntrico.

Ejecución del programa

```
Elementos del array
Elemento [0] = 1
Elemento [1] = 2
Elemento [2] = 3
Elemento [3] = 3
Elemento [4] = 4
Elemento [5] = 4
Elemento [6] = 5
Elemento [7] = 2
Elemento duplicado: 2
Elemento duplicado: 3
Elemento duplicado: 4
```

Figura 1.11. Ejecución del programa del ejercicio 1.5.

Ejercicios propuestos

- Desarrollar un programa que determine el elemento mayor y menor de un *array* de enteros.
- Desarrollar un programa que, dado un número entero, busque dicho número en un *array*.
- Desarrollar un programa que busque elementos comunes en dos *arrays* de enteros.