

4 | Programación estructurada

Meta

Que el alumno diseñe algoritmos imperativos estructurados para resolver una familia de problemas y los implemente.

Objetivos

Al finalizar la práctica el alumno será capaz de:

1. Leer diagramas de Warnier-Orr para analizar un problema en forma estructurada.
2. Plantear un problema en forma iterativa utilizando variables y estructuras de control.
3. Programar funciones usando estructuras de control en Java.

Antecedentes

Funciones

En la programación estructurada, las funciones son unidades de código que reciben argumentos, ejecutan un algoritmo y devuelven la respuesta a un problema.

Los argumentos son datos de un tipo dado, al igual que los resultados. Para definir las entradas y salidas de una función se utilizan los encabezados.

Ejemplo 4.1 *El encabezado de una función que calcule el factorial de un número en Java se puede ver así:*

```
1 public static long factorial(long n)
```

Para *declarar* una función en Java se utiliza la siguiente estructura:

```
<acceso>           ::= public | private | protected | {}
<modificador>      ::= final | static | {}
<identificador>    ::= (<letra> | _) (<letra> | <dígito> | _)*

<parámetro>        ::= <tipo> <identificador>
<parámetros>       ::= <parámetro> (, <parámetros>)* | {}

<función>          ::=
<acceso> <modificador> <tipo> <identificador> (<parámetros>) {
    <implementación>
}
```

El *acceso* sirve para determinar quiénes pueden ver y usar la función. De los modificadores, sólo usamos *static* por el momento, y nos sirve para indicar que la función hace su trabajo por sí sola, sin necesidad de recurrir a información fuera de los parámetros que recibe.

De este modo, cuando se quiere la respuesta a un problema, no es necesario conocer los detalles de cómo fue resuelto, sino sólo invocar a la función que lo resuelve.

Ejemplo 4.2 *Para conocer el factorial de un número, basta con llamar a la función, sin necesidad de saber cómo se le calculó:*

```
1 long f = factorial(5);
```

A esta propiedad que permite utilizar el código sin necesidad de conocer los detalles de una solución, se le denomina *encapsulamiento*¹.

Polimorfismo: Sobrecarga

Cabe mencionar que Java permite definir funciones con el mismo nombre, siempre y cuando los tipos de los argumentos sean distintos. Conceptualmente esto permite hablar de que *se ejecuta una misma función*, que reacciona diferente dependiendo del tipo de sus argumentos. A esta característica se le llama *polimorfismo*.

Ejemplo 4.3 *La función `imprime` de la clase `ImpresoraBinario`, que utilizaste en la práctica de tipos primitivos y bits tiene cuatro definiciones distintas, dependiendo del tipo de dato cuyos bits se quieren imprimir en pantalla:*

¹El término encapsulamiento surge con la programación orientada a objetos, sin embargo, su definición aplica igualmente a la forma en que usamos las funciones en la programación estructurada.

```

1 public void imprime(int num)
2 public void imprime(long num)
3 public void imprime(float num)
4 public void imprime(double num)

```

Lo que distingue a una implementación de otra no está determinado por todo el encabezado si no sólo por lo que se conoce como la *firma* del método, que consta del nombre del método y la lista de tipos de sus parámetros.

```

<tipos> ::= <tipo>(,<tipo>)*|{}
<firma> ::= <identificador>(<tipos>)

```

Ejemplo 4.4 Las firmas de los métodos *imprime* son:

```

1 imprime(int)
2 imprime(long)
3 imprime(float)
4 imprime(double)

```

Solamente es posible definir métodos con el mismo nombre, si las firmas son distintas. A esto se le llama *sobrecargar* el método.

Warnier-Orr

Dentro de la función se implementa el algoritmo mediante órdenes, que la computadora debe ir ejecutando en orden. Esta forma de programar hace que este paradigma de programación reciba el nombre de *imperativo*.

Existen cuatro formas de organizar las secuencias de órdenes en la programación estructurada: secuencial, condicional, iterativa y, en los lenguajes modernos, recursiva.

Para diseñar un algoritmo utilizando este ordenamiento de las instrucciones, se cuenta con la ayuda de la metodología de Warnier-Orr (**Viso2012**). En esta metodología pondremos especial atención en la forma en que recibimos, procesamos y producimos los datos que requiere nuestra función para devolver su resultado. Se analiza el problema sin escribir código, de modo que la solución sea independiente del lenguaje concreto que vayamos a utilizar después. Los símbolos y frases casi asemejan a instrucciones de código, pero van mezcladas con símbolos matemáticos y textos que esbozan los procedimientos a realizar, por ello se le llama *pseudocódigo*.

Secuencial

Cuando los datos se deben procesar en secuencia.

$$\text{nombre} = \begin{cases} \text{descr}_1 \\ \text{descr}_2 \\ \dots \\ \text{descr}_n \end{cases} \quad (4.1)$$

La manera en la que se interpreta el diagrama exterior es: El programa realiza las acciones descritas en $\text{descr}_1, \text{descr}_2, \dots, \text{descr}_n$ de manera secuencial y al finalizar el programa termina.

Condicional

Cuando la forma de continuar el proceso depende del valor de los datos.

$$\text{nombre} = \begin{cases} \text{cond}_1 & \begin{cases} \dots \\ \dots \end{cases} \\ \oplus \\ \text{cond}_2 & \begin{cases} \dots \\ \dots \end{cases} \\ \oplus \\ \dots \\ \oplus \\ \text{cond}_n & \begin{cases} \dots \\ \dots \end{cases} \end{cases} \quad (4.2)$$

La manera en la que se interpreta el diagrama exterior es: Si se cumple cond_1 , entonces el programa realiza las acciones descritas en descr_1 . Al finalizar esto, se hace lo análogo con cada condición $\text{cond}_2, \dots, \text{cond}_n$ de manera secuencial. El programa termina tras haber revisado todas las condiciones y haber ejecutado lo correspondiente a cada condición que se cumplió.

Iterativa

Cuando repetir el mismo conjunto de instrucciones permite evolucionar a los datos, hasta llevarlos al resultado deseado.

$$\text{nombre} \begin{matrix} \text{(condición)} \end{matrix} = \begin{cases} \text{descr}_1 \\ \text{descr}_2 \\ \dots \\ \text{descr}_n \end{cases} \quad (4.3)$$

El diagrama anterior se interpreta de la siguiente manera: El programa revisa si condición se cumple. Si no se cumple el programa termina. Si sí se cumple, el programa ejecuta $\text{descr}_1, \text{descr}_2, \dots, \text{descr}_n$ de manera secuencial y al terminar vuelve a realizar todo lo descrito anteriormente, desde el punto en que revisa si condición se cumple.

Recursiva

Cuando el proceso está definido en términos de sí mismo, pero se aplica sobre los datos modificados.

$$\text{nombre} = \begin{cases} \text{descr}_1 & \text{caso base} \\ f(\text{nombre}) & \text{llamada recursiva} \end{cases} \quad (4.4)$$

Esquema de una solución

El esquema completo para la solución de un problema se ve:

$$\text{Nombre del problema} = \begin{cases} \text{.Principio} & \begin{cases} \text{Obtener datos} \\ \text{Inicializar} \\ \dots \end{cases} \\ \text{Proceso} & \begin{cases} \dots \\ \dots \end{cases} \\ \text{.Final} & \begin{cases} \text{Amarrar cabos sueltos} \\ \text{Entregar resultados} \end{cases} \end{cases} \quad (4.5)$$

Ejemplo 4.5 *La definición matemática de factorial dice que:*

$$\text{factorial}(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * \text{factorial}(n - 1) & \text{si } n > 0 \end{cases} \quad (4.6)$$

Lo cual solemos desarrollar como:

$$n! = n(n-1)(n-2)\dots 1 \quad (4.7)$$

Entonces podemos calcular el factorial de un número multiplicando a todos los naturales menores o iguales a él.

$$\text{Factorial} = \left\{ \begin{array}{l} \text{.Principio} \\ \text{fact} \\ (n > 1) \\ \text{.Final} \end{array} \right. \left\{ \begin{array}{l} n \leftarrow \\ \begin{cases} \text{¿}n \geq 0? \checkmark \\ f \leftarrow 1 \end{cases} \\ = \begin{cases} f \leftarrow f * n \\ n \leftarrow n - 1 \end{cases} \\ \text{Devuelve } f \end{array} \right. \quad (4.8)$$

Aquí observamos que el caso $n = 0$ y de paso $n = 1$ se cumplen desde el .Principio.

Contratos

Para que un algoritmo funcione, los datos que recibe como entrada deben cumplir ciertas condiciones. Por ejemplo, no podemos calcular el factorial de un número negativo. Estas condiciones son las *precondiciones* del algoritmo y en el código se refuerzan de dos maneras:

- En la *documentación* de la función, donde se especifica lo que hace y los requisitos en los datos para que ésta funcione correctamente. A esto se le llama el *contrato* de la función, pues exige se cumplan las precondiciones y se compromete a cumplir con las poscondiciones.
- Mediante una verificación de que la precondiciones se hayan cumplido al inicio de la ejecución de la función. Si estas precondiciones no se han cumplido, el código lanza un mensaje indicando la precondición que se violó.

Implementación

Una vez analizado el problema y diseñado el algoritmo para resolverlo, procedemos a traducir esa solución a un lenguaje de programación, en este caso a Java.

Cada forma de organizar las instrucciones del paradigma estructurado se implementan utilizando estructuras de control. En Java la sintaxis (forma de escribir) y semántica (el significado) utilizadas son:

Secuencial

Se escribe un enunciado seguido de otro. Cada enunciado puede ser simple y terminar en punto y coma, o compuesto y contener un bloque de instrucciones. Este es el flujo

normal de un programa imperativo, así que no necesita de una estructura de control.

```

<enunciado-simple> ::= <comando>;
<bloque>           ::= {
                        <secuencia>
                      }
<enunciado>        ::= <enunciado-simple> | <bloque>
<secuencia>        ::= <enunciado>*

```

Condicional

Hay dos formas en Java, el `if then else` y el `switch`.

```

<if>      ::= if (<condición>)
              <secuencia>
              (<else> | ∅)
<else>    ::= else <secuencia>

```

La condición es cualquier expresión booleana, devuelve `true` o `false`. La secuencia que prosigue al `if` solamente se ejecuta si la condición es verdadera. Ejecutar algún código alternativo si la condición es falsa es opcional y se indica mediante el `else`. Es posible encadenar varias cláusulas `if` de modo siguiente:

```

1  if(x == 0) {
2    // Haz algo
3  } else if (x > 0) {
4    // Haz otra cosa
5  } else {
6    // El último recurso
7  }

```

El `switch` se utiliza cuando hay varias opciones. El código en su interior se ejecuta a partir del `case` correspondiente al valor de la variable y hasta que termine el bloque del `switch` o un `break` de por terminada su ejecución.

```

<switch> ::= switch(<variable>) {
              (<case>)*
              (<default>|∅)
            }
<case>   ::= case <caso>:
              <secuencia>
              (break;|∅)

```

Los casos se indican con enteros o caracteres y `break` termina la ejecución dentro del `switch`.

Iterativa

Java tiene tres tipos de ciclos: `while`, `do while` y `for`.

```
<while> ::= while(<condición>)
           <secuencia>
<do>     ::= do
           <secuencia>
           while(<condición>)
<for>    ::= for(<inicio>; <condición>; <actualización>)
           <secuencia>
```

La principal diferencia entre `while` y `do`, es que el primero podría nunca ejecutarse, mientras que está garantizado que el segundo siempre se ejecutará al menos una vez. En el caso del `for`, `<inicio>` sólo se ejecuta una vez, antes de ejecutar el ciclo por primera vez y `<actualización>` siempre, al final de cada ciclo.

Los tres ciclos son afectados por `break`, que interrumpe la ejecución del ciclo y `continue`, que se brinca el resto de las instrucciones y continua ejecutando el ciclo pero en la siguiente iteración.

Ejemplo 4.6 El código siguiente muestra cómo se calcula el factorial de un número en forma iterativa, programado a partir del diagrama de Warnier-Orr anterior.

```
1 public class Ciclos {
2     /**
3      * Calcula el factorial de un número.
4      * @param n el número cuyo factorial se quiere calcular.
5      * @throws ArithmeticException si n < 0.
6      */
7     public static long factorial(long n) {
8         if (n < 0) throw new ArithmeticException
9             ("No está definido el factorial de un número negativo");
10        long f = 1;
11        while(n > 1) {
12            f *= n;
13            n--;
14        }
15        return f;
16    }
17
18    public static void main(String[] args) {
19        long n = Long.parseLong(args[0]);
20        long f = factorial(n);
21        System.out.println("El factorial de " + n + " es " + f);
22    }
23 }
```


Desarrollo

En esta práctica se modificarán los archivos de la práctica anterior sobre el compilador para crear un programa que reciba un entero positivo como parámetro e imprima en la pantalla si es un número primo o no (se te proporcionan todos los archivos que necesitarás modificar en tu repositorio, no tienes que agregar ningún otro archivo de código).

¿Qué necesitas para determinar si un número es primo o no? Es verdad que la forma más común es intentar dividirlo entre los primos más chicos que él, pero por ahora no tenemos de dónde sacar una lista de números primos, así que hay que comenzar por lo más sencillo, la definición de número primo:

Definición 4.1: Número primo

Un número es primo si tiene exactamente dos divisores 1 y él mismo.

Siguiendo esta definición, se muestra a continuación una parte de un diagrama de Warnier-Orr que inicia el análisis de este problema. Complétalo para generar tu código.

$$\text{Nombre del problema} = \left\{ \begin{array}{ll} \text{.Principio} & \left\{ \begin{array}{l} n \leftarrow \\ i \leftarrow 2 \\ \text{divisible} \leftarrow \text{no} \end{array} \right. \\ \text{divisores} & = \left\{ \begin{array}{l} i \leftarrow \text{siguiente candidato} \\ \text{es divisible} \{ \\ \oplus \\ \text{no es divisible} \{ \\ \text{.Final} & \left\{ \text{devolver } \neg \text{divisible} \end{array} \right. \end{array} \right. \quad (4.9)$$

Ejercicios

1. Cambia el nombre de la clase Entrada a EsPrimo para que refleje el funcionamiento de tu programa. También cambia el paquete de tu proyecto, para que ahora se llame `icc.funciones`. Modifica los nombres de los directorios y archivos, y al `build.xml` para que compile tu nuevo proyecto.
2. Modifica el método `main` para leer el único argumento. Usa la función `Inte-`

`ger.parseInt()`² para obtener el entero.

```
1 int n = Integer.parseInt(args[0]);
```

3. Agrega una función estática llamada `esPrimo` que reciba un entero como argumento y devuelva un boolean indicando si el número es primo o no.
4. Documenta la función que programaste.
5. Manda llamar la función desde el método `main`, pasando como argumento el número indicado por el usuario.

Entregables

Deberás actualizar en tu repositorio con las modificaciones especificadas en la sección de **Ejercicios**. También deberás modificar y completar el diagrama de Warnier-Orr de la sección **Desarrollo** y añadirlo en algún archivo a tu repositorio.

²`parseInt` es una función estática programada en la clase `Integer` del API de Java.