3 | 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`.

Código Auxiliar 3.1: Programación estructurada

https://github.com/computacion-ciencias/icc-programacion-estructurada

Antecedentes

Definición 3.1: Programación estructurada

"El *diseño estructurado* es un conjunto de consideraciones generales de diseño de programas y técnicas para que programar, depurar y modificar programas sea más fácil, rápido y menos caro. El proceso involucra programar en módulos manejables y definibles. El término *módulo* se utiliza para referirse a un conjunto de uno o más enunciados contiguos de un programa identificables por un nombre mediante el

cual otras partes del sistema pueden invocarlo y preferentemente teniendo su propio conjunto distintivo de nombres de variables." (Erbschloe 2021)

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* (también llamados *parámetros*) 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 3.1. *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)
```

El acceso `public` indica que esta función puede ser llamada en cualquier programa; usamos `static` para indicar que la función hace su trabajo por sí sola, sin necesidad de recurrir a información fuera de los parámetros que recibe; `long` indica que el resultado que devuelve es un número de entero de 64 bits; `factorial` es el nombre de la función y `long n` indica que la función requiere recibir como argumento un entero de 64 bits, al cual llamaremos `n`.

En general, 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.

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

```
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 fución*, que reacciona diferente dependiendo del tipo de sus argumentos. A esta característica se le llama *polimorfismo*.

Ejemplo 3.3. *La función* `imprime` *de la clase* `ImpresoraBinario`*, que utilizaste en el Capítulo 2, tiene cuatro definiciones distitas, dependiendo del tipo de dato cuyos bits se quieren imprimir en pantalla:*

```
public void imprime(int num)
public void imprime(long num)
public void imprime(float num)
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 (*Defining Methods* s.f.).

```
<tipos> ::= <tipo>(,<tipo>)*|∅
<firma> ::= <identificador>(<tipos>)
```

Ejemplo 3.4. *Las firmas de los métodos* `imprime` *son:*

```
imprime(int)
imprime(long)
imprime(float)
imprime(double)
```

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

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

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 (Viso y Peláez 2012, págs. 52-60). 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.

$$\text{nombre} \begin{cases} \text{descr}_1 \\ \text{descr}_2 \\ \dots \\ \text{descr}_n \end{cases} \tag{3.1}$$

Condicional

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

$$\text{nombre} \begin{cases} \text{cond}_1 & \begin{cases} \dots \\ \dots \end{cases} \\ \oplus \\ \text{cond}_2 & \begin{cases} \dots \\ \dots \end{cases} \\ \oplus \\ \dots \\ \oplus \\ \text{cond}_n & \begin{cases} \dots \\ \dots \end{cases} \end{cases} \tag{3.2}$$

Iterativa

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

$$\begin{matrix}\mathtt{nombre}\\(\mathtt{condición})\end{matrix}\begin{cases}\mathtt{descr}_1\\\mathtt{descr}_2\\...\\\mathtt{descr}_n\end{cases} \tag{3.3}$$

Recursiva

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

$$\mathtt{nombre} = \begin{cases}\mathtt{descr}_1 & \text{caso base}\\ \mathtt{f}(\mathtt{nombre}) & \text{llamada recursiva}\end{cases} \tag{3.4}$$

donde $\mathtt{f}(\mathtt{nombre})$ indica algún algoritmo que vuelve a invocar a la misma función, pero sobre argumentos diferentes.

Esquema de una solución

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

$$\text{Nombre del problema} = \begin{cases}.\mathtt{Principio} & \begin{cases}\text{Obtener datos}\\\text{Inicializar}\end{cases}\\ \mathtt{Proceso} & \begin{cases}...\\\\...\end{cases}\\ .\mathtt{Final} & \begin{cases}\text{Amarrar cabos sueltos}\\\text{Entregar resultados}\end{cases}\end{cases} \tag{3.5}$$

Ejemplo 3.5. *La definición matemática de factorial dice que:*

$$\mathtt{factorial}(n) = \begin{cases}1 & \textit{si } n = 0\\ n * \mathtt{factorial}(n-1) & \textit{si } n > 0\end{cases} \tag{3.6}$$

Lo cual solemos desarrollar como:

$$n! = n(n-1)(n-2)...1 \tag{3.7}$$

Entonces podemos calcular el factorial de un número multiplicando a todos los naturales menores o iguales a él. Utilizaremos una flecha apuntando hacia el nombre de un variable ← para indicar que se le asignará el valor a su derecha, o simplemente que se recibirá ese valor. Como estas descripciones no son formales, también podemos usar símbolos como una palomita (✓) para indicar que se cumple alguna condición.

$$\textit{Factorial} = \begin{cases} .\texttt{Principio} & \begin{cases} n \leftarrow \\ \text{¿}n \geqslant 0\text{?} \checkmark \\ f \leftarrow 1 \end{cases} \\ \texttt{fact} \\ (n > 1) & \begin{cases} f \leftarrow f * n \\ n \leftarrow n - 1 \end{cases} \\ .\texttt{Final} & \begin{cases} \textit{Devuelve } f \end{cases} \end{cases} \tag{3.8}$$

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

El objetivo de esta metodología es ayudar en el proceso de diseño de una solución, por lo que se puede utilizar y modificar con mucha más libertad que un lenguaje formal.

Contratos

Para que un algortimo 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 las 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 constar de un bloque de instrucciones.

```
<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>)
             <enunciado>
           (<else> | ∅)
<else> ::= else <enunciado>
```

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

```
<switch> ::= switch(<variable>) {
               (<case>)*
               (<default>|∅)
             }
```

```
<case>   ::= case <caso>:
               <secuencia>
               (break;|∅)
```

El código en su interior se ejecuta a partir del `case` correspondiente al valor (`<caso>`) de la variable y hasta que termine el bloque del `switch` o un `break` de por terminada su ejecución. 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`. Su contenido se ejecuta mientras la condición que se establezca se cumpla, es decir, la expresión que se use debe dar como resultado `true`.

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

La principal diferencia entre `while` y `do`, es que el contenido del primero podría nunca ejecutarse si la condición es falsa desde un principio, 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 3.6. *El código siguiente muestra cómo se calcula el factorial de un número en forma iterativa, programado a partir del diagrama de Warnier-Orr anterior. La función `factorial` implementa el algoritmo y la función `main` la invoca con el valor que le pase el usuario del programa al invocarlo.*

Listado 3.1: Ciclos.java

```
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    /** Punto de entrada, desde aquí se ejecuta el programa. */
19    public static void main(String[] args) {
20      long n = Long.parseLong(args[0]);  // Lee el argumento pasado al
          ↪ programa.
21      long f = factorial(n);
22      System.out.println("El factorial de " + n + " es " + f);
23    }
24  }
```

Desarrollo

En esta práctica se modificará el código auxiliar que se te entrega, 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 intenter 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 3.2: 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.

$$\text{Es primo}\begin{cases}\text{.Principio} & \begin{cases} n \leftarrow \\ i \leftarrow 2 \\ divisible \leftarrow no \end{cases} \\ \begin{matrix} divisores \\ (i <?) \end{matrix} & \begin{cases} \text{es divisible } \{ \\ \oplus \\ \text{no es divisible } \{ \\ i \leftarrow \text{siguiente candidato} \end{cases} \\ \text{.Final} & \{\text{devolver } \neg divisible \end{cases} \tag{3.9}$$

1. Cambia el nombre de la clase `Entrada` a `DetectorDePrimos` 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 al `build.xml` para que compile tu nuevo proyecto.

2. Modifica el método `main` para leer el único argumento. Usa la función Integer.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 tu repositorio con las modificaciones especificadas en la sección de **Ejercicios**. También deberás modificar y completar el diagrama de Warnier-Orr de la Sección 3.4 y añadirlo en algún archivo a tu repositorio.

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