

# Cómo pensar como un informático

Allen B. Downey  
Ricardo Pérez López

Versión Eiffel, Primera Edición  
(Alfa)

# Cómo pensar como un informático

Versión Eiffel, Primera Edición (Alfa)

Copyright (C) 2000 Allen B. Downey

Copyright (C) 2002 de la traducción y adaptación, Ricardo Pérez López

This book is an Open Source Textbook (OST). Permission is granted to reproduce, store or transmit the text of this book by any means, electrical, mechanical, or biological, in accordance with the terms of the GNU General Public License as published by the Free Software Foundation (version 2).

This book is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

The original form of this book is  $\text{\LaTeX}$  source code. Compiling this  $\text{\LaTeX}$  source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed. All intermediate representations (including DVI and Postscript), and all printed copies of the textbook are also covered by the GNU General Public License.

The  $\text{\LaTeX}$  source for this book, and more information about the Open Source Textbook project, is available from

<http://rocky.wellesley.edu/downey/ost>

or by writing to Allen B. Downey, Computer Science Dept, Wellesley College, Wellesley, MA 02482.

El formato original de la versión traducida y adaptada a Eiffel se encuentra en código fuente  $\text{\LaTeX}$ . A partir de este código fuente se puede generar una representación del libro independiente del dispositivo, que puede luego ser convertido a otros formatos y ser impreso. Todas las representaciones intermedias (incluyendo DVI y PostScript), y todas las copias impresas del libro también están cubiertas por la Licencia Pública General GNU.

El código fuente de esta traducción y adaptación puede obtenerse escribiendo un correo electrónico a Ricardo Pérez <[ricpelo@eresmas.com](mailto:ricpelo@eresmas.com)>.

La Licencia Pública General GNU está disponible en <http://www.gnu.org> o escribiendo a la Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

This book was typeset by the author using  $\text{\LaTeX}$  and dvips, which are both free, open-source programs. Su traducción fue escrita usando  $\text{\LaTeX}$  y dvips, que también son programas open-source.

# Índice general

<b>1. El camino del programa</b>	<b>1</b>
1.1. ¿Qué es un lenguaje de programación?	1
1.2. ¿Qué es un programa?	2
1.3. ¿Qué es depurar?	3
1.4. Lenguajes formales y naturales	5
1.5. El primer programa	6
1.6. Glosario	7
<b>2. Variables y tipos</b>	<b>9</b>
2.1. Más impresión	9
2.2. Variables	10
2.3. Asignación	11
2.4. Imprimir variables	12
2.5. Palabras clave	14
2.6. Operadores	14
2.7. Orden de las operaciones	15
2.8. Operadores sobre cadenas (STRINGS)	15
2.9. Composición	15
2.10. Glosario	16
<b>3. Métodos</b>	<b>18</b>
3.1. Reales	18
3.2. Convertir un DOUBLE en un INTEGER	19
3.3. Métodos matemáticos	21
3.4. Composición	22
3.5. Añadir nuevos métodos	23
3.6. Clases y métodos	25
3.7. Programas con varios métodos	26
3.8. Parámetros y argumentos	26
3.9. Métodos con varios parámetros	28
3.10. Diagramas de pila	29
3.11. Métodos con resultados	29
3.12. Glosario	30

<b>4. Alternativas y recursividad</b>	<b>31</b>
4.1. Los operadores módulo y división entera . . . . .	31
4.2. Ejecución alternativa . . . . .	31
4.3. Ejecución alternativa doble . . . . .	32
4.4. Alternativas encadenadas . . . . .	33
4.5. Alternativas anidadas . . . . .	33
4.6. Recursividad . . . . .	34
4.7. Diagramas de pila para métodos recursivos . . . . .	35
4.8. Glosario . . . . .	36
<b>5. Métodos función</b>	<b>37</b>
5.1. Valores de retorno . . . . .	37
5.2. Desarrollo de programas . . . . .	39
5.3. Composición . . . . .	41
5.4. Expresiones lógicas . . . . .	41
5.5. Operadores lógicos . . . . .	42
5.6. Métodos lógicos . . . . .	43
5.7. Más recursividad . . . . .	43
5.8. Salto de fe . . . . .	45
5.9. Un ejemplo más . . . . .	46
5.10. Glosario . . . . .	47
<b>6. Iteración</b>	<b>48</b>
6.1. Varias asignaciones . . . . .	48
6.2. Iteración . . . . .	48
6.3. La sentencia <code>loop</code> . . . . .	49
6.4. Tablas . . . . .	50
6.5. Tablas bidimensionales . . . . .	53
6.6. Encapsulación y generalización . . . . .	54
6.7. Métodos . . . . .	55
6.8. Más encapsulación . . . . .	55
6.9. Variables locales . . . . .	56
6.10. Más generalización . . . . .	56
6.11. Glosario . . . . .	58
<b>7. Cadenas y cosas</b>	<b>59</b>
7.1. Invocando métodos sobre objetos . . . . .	59
7.2. Longitud . . . . .	60
7.3. Recorrido . . . . .	61
7.4. Errores en tiempo de ejecución . . . . .	61
7.5. Leer la documentación . . . . .	62

7.6.	El método <code>first_index_of</code> . . . . .	62
7.7.	Diseño por Contrato . . . . .	63
7.8.	Iterar y contar . . . . .	64
7.9.	Caracteres y números . . . . .	65
7.10.	Las cadenas cambian . . . . .	66
7.11.	Las cadenas son incomparables . . . . .	67
7.12.	Glosario . . . . .	68
<b>8.</b>	<b>Objetos interesantes</b>	<b>69</b>
8.1.	¿Qué es interesante? . . . . .	69
8.2.	Objetos <code>PUNTO</code> . . . . .	70
8.3.	Atributos . . . . .	71
8.4.	El Principio del Acceso Uniforme . . . . .	72
8.5.	Objetos como parámetros . . . . .	72
8.6.	Rectángulos . . . . .	73
8.7.	Objetos como tipos de retorno . . . . .	73
8.8.	Los objetos cambian . . . . .	74
8.9.	Alias . . . . .	77
8.10.	<code>Void</code> . . . . .	78
8.11.	Recogida de basura . . . . .	79
8.12.	Referencias y objetos expandidos . . . . .	79
8.13.	Glosario . . . . .	80
<b>9.</b>	<b>Crear tus propios objetos</b>	<b>81</b>
9.1.	Definiciones de clases y tipos de objetos . . . . .	81
9.2.	Hora . . . . .	82
9.3.	Constructores . . . . .	83
9.4.	Más constructores . . . . .	83
9.5.	La variable especial <code>Current</code> . . . . .	84
9.6.	Crear un nuevo objeto . . . . .	85
9.7.	Imprimir un objeto . . . . .	86
9.8.	Operaciones sobre objetos . . . . .	87
9.9.	Funciones puras . . . . .	87
9.10.	Modificadores . . . . .	89
9.11.	Métodos rellenadores . . . . .	90
9.12.	¿Cuál es mejor? . . . . .	91
9.13.	Desarrollo incremental contra planificación . . . . .	91
9.14.	Generalización . . . . .	92
9.15.	Algoritmos . . . . .	93
9.16.	Glosario . . . . .	93

<b>10.Arrays</b>	<b>95</b>
10.1. Acceder a los elementos . . . . .	96
10.2. <code>lower</code> , <code>upper</code> y <code>count</code> . . . . .	97
10.3. Copiar arrays . . . . .	97
10.4. Números aleatorios . . . . .	99
10.5. Estadísticas . . . . .	99
10.6. Array de números aleatorios . . . . .	100
10.7. Contando . . . . .	101
10.8. Muchos trozos . . . . .	102
10.9. Una solución de un sólo paso . . . . .	104
10.10. Glosario . . . . .	105
<b>11.Arrays de objetos</b>	<b>106</b>
11.1. Composición . . . . .	106
11.2. Objetos <code>CARTA</code> . . . . .	106
11.3. El método <code>imprimir_carta</code> . . . . .	108
11.4. El método <code>misma_carta</code> . . . . .	109
11.5. El método <code>comparar_carta</code> . . . . .	110
11.6. Arrays de cartas . . . . .	112
11.7. El método <code>imprimir_baraja</code> . . . . .	113
11.8. Búsqueda . . . . .	114
11.9. Barajas y sub-barajas . . . . .	117
11.10. Glosario . . . . .	118
<b>12.Objetos de Arrays</b>	<b>119</b>
12.1. Barajar . . . . .	121
12.2. Ordenar . . . . .	122
12.3. Sub-barajas . . . . .	122
12.4. Barajar y dar . . . . .	124
12.5. Ordenación por mezcla . . . . .	124
12.6. Glosario . . . . .	126
<b>13.Programación orientada a objetos</b>	<b>127</b>
13.1. Lenguajes y estilos de programación . . . . .	127
13.2. Métodos de objeto . . . . .	127
13.3. El objeto actual . . . . .	128
13.4. Números complejos . . . . .	128
13.5. Una función sobre números <code>COMPLEJOS</code> . . . . .	129
13.6. Otra función sobre números <code>COMPLEJOS</code> . . . . .	130
13.7. Un nombre más adecuado . . . . .	131
13.8. Un modificador . . . . .	131

13.9. El método <code>to_string</code> . . . . .	132
13.10. El método <code>is_equal</code> . . . . .	134
13.11. Invocar a un método de objeto desde otro . . . . .	135
13.12. Herencia . . . . .	135
13.13. Rectángulos dibujables . . . . .	136
13.14. La jerarquía de clases . . . . .	137
13.15. Diseño orientado a objetos . . . . .	137
13.16. Glosario . . . . .	138
<b>14.Listas enlazadas</b>	<b>139</b>
14.1. Referencias en objetos . . . . .	139
14.2. La clase <code>NODO</code> . . . . .	139
14.3. Listas como colecciones . . . . .	141
14.4. Listas y recursividad . . . . .	142
14.5. Listas infinitas . . . . .	143
14.6. El teorema fundamental de la ambigüedad . . . . .	144
14.7. Otros métodos para nodos . . . . .	145
14.8. Modificar listas . . . . .	145
14.9. Envoltorios y ayudantes . . . . .	146
14.10. La clase <code>LISTA_ENLAZADA</code> . . . . .	147
14.11. Invariantes . . . . .	148
14.12. Invariantes en Eiffel . . . . .	149
14.13. Glosario . . . . .	150
<b>15.Pilas</b>	<b>151</b>
15.1. Tipos abstractos de datos . . . . .	151
15.2. El TAD Pila . . . . .	151
15.3. Genericidad . . . . .	152
15.4. El objeto <code>DS_LINKED_STACK</code> . . . . .	152
15.5. Expresiones postfijas . . . . .	154
15.6. Implementar TADs . . . . .	155
15.7. Más genericidad . . . . .	155
15.8. Implementación del TAD Pila basada en array . . . . .	156
15.9. Cambiar el tamaño de los arrays . . . . .	158
15.10. Glosario . . . . .	159

<b>16. Colas</b>	<b>160</b>
16.1. El TAD Cola . . . . .	160
16.2. Envoltorio . . . . .	161
16.3. Cola enlazada . . . . .	163
16.4. Buffer circular . . . . .	165
16.5. Cola con prioridad . . . . .	168
16.6. Clase diferida . . . . .	169
16.7. Implementación de Cola con prioridad basada en array . . . . .	169
16.8. Un cliente de Cola con prioridad . . . . .	171
16.9. La clase <b>GOLFISTA</b> . . . . .	172
16.10. Glosario . . . . .	175
<b>17. Árboles</b>	<b>177</b>
17.1. Un nodo del árbol . . . . .	177
17.2. Construir árboles . . . . .	178
17.3. Recorrer árboles . . . . .	179
17.4. Árboles de expresiones . . . . .	180
17.5. Recorrido . . . . .	180
17.6. Encapsulación . . . . .	182
17.7. Definir una clase diferida . . . . .	182
17.8. Hacer efectiva una clase diferida . . . . .	183
17.9. Implementación de árboles basada en array . . . . .	184
17.10. Diseño por Contrato . . . . .	188
17.11. La clase <b>ARRAY</b> . . . . .	189
17.12. La clase <b>ITERATOR</b> . . . . .	190
17.13. Glosario . . . . .	191
<b>18. Montículos</b>	<b>192</b>
18.1. El Montículo . . . . .	192
18.2. Análisis de eficiencia . . . . .	192
18.3. Análisis de la ordenación por mezcla . . . . .	194
18.4. Sobrecarga . . . . .	196
18.5. Implementaciones de Colas con prioridad . . . . .	196
18.6. Definición de Montículo . . . . .	197
18.7. Eliminar de un montículo . . . . .	198
18.8. <b>insertar</b> en un montículo . . . . .	200
18.9. Eficiencia de los montículos . . . . .	200
18.10. Heapsort . . . . .	201
18.11. Glosario . . . . .	202



<b>19. Tablas</b>	<b>203</b>
19.1. Arrays y Tablas . . . . .	203
19.2. El TAD Tabla . . . . .	203
19.3. La clase DS_HASH_TABLE . . . . .	204
19.4. Implementación basada en array . . . . .	206
19.5. La clase diferida COLLECTION . . . . .	208
19.6. Implementación de tabla hash . . . . .	208
19.7. Funciones hash . . . . .	209
19.8. Redimensionar una tabla hash . . . . .	210
19.9. Eficiencia de la redimensión . . . . .	211
19.10. Glosario . . . . .	211

# Capítulo 1

## El camino del programa

El objetivo de este libro, y esta clase, es enseñarte a pensar como un informático. Me gusta la forma en la que piensan los informáticos porque combinan algunas de las mejores características de las Matemáticas, la Ingeniería y las Ciencias Naturales. Como los matemáticos, los informáticos usan lenguajes formales para representar ideas (más específicamente, cálculos o computaciones). Como los ingenieros, diseñan cosas, ensamblan componentes para formar sistemas y evalúan alternativas. Como los científicos, observan el comportamiento de sistemas complejos, formulan hipótesis, y comprueban predicciones.

La habilidad más importante de un informático es la capacidad de **resolver problemas**. Por tal me refiero a la capacidad para formular problemas, pensar en soluciones de manera creativa, y expresar una solución de forma clara y precisa. Como podrá apreciarse, el proceso de aprender a programar es una oportunidad excelente para practicar la capacidad de resolución de problemas. Es por ello por lo que este capítulo se llama “El camino del programa”.

A un cierto nivel, aprenderás a programar, lo cual es una habilidad útil por sí misma. A otro nivel llegarás a usar la programación como medio para llegar a un fin. A medida que vayamos avanzando, el fin se hará más y más nítido.

### 1.1. ¿Qué es un lenguaje de programación?

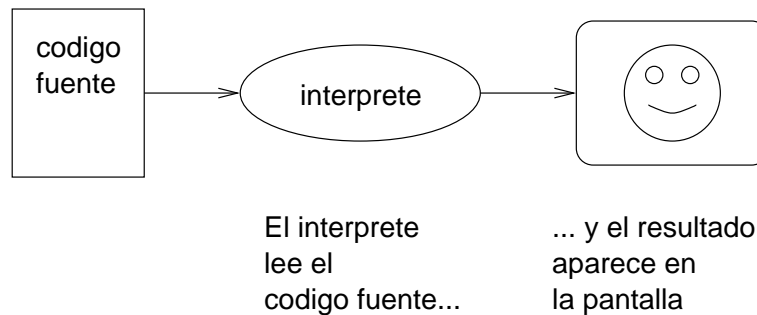
El lenguaje de programación que vas a aprender es Eiffel, el cual ha ido evolucionando progresivamente desde su aparición inicial en 1985. Eiffel es un ejemplo de **lenguaje de alto nivel**; otros lenguajes de alto nivel de los que puede que hayas oído hablar son Pascal, C, C++, FORTRAN y Java.

Como habrás deducido del nombre “lenguaje de alto nivel”, existen además los **lenguajes de bajo nivel**, a los que a veces se refieren como lenguaje máquina o lenguaje ensamblador. Coloquialmente hablando, los ordenadores sólo pueden ejecutar programas escritos en lenguajes de bajo nivel. Por consiguiente, los programas escritos en un lenguaje de alto nivel deben ser traducidos antes de que puedan ejecutarse. Esa traducción lleva cierto tiempo, lo que es una pequeña desventaja de los lenguajes de alto nivel.

Pero las ventajas son enormes. En primer lugar, es *mucho* más fácil programar en un lenguaje de alto nivel; por “más fácil” me refiero a que el programa tarda menos tiempo en ser escrito, es más corto y más fácil de leer, y es más probable que sea correcto. En segundo lugar, los lenguajes de alto nivel son **transportables**, lo que significa que pueden ejecutarse en diferentes tipos de ordenadores con pocas o ninguna modificación. Los programas escritos en lenguajes de bajo nivel sólo pueden ejecutarse en un cierto tipo de ordenador, y deben ser reescritos para ejecutarse en otro.

Debido a esas ventajas, casi todos los programas están escritos en lenguajes de alto nivel. Los lenguajes de bajo nivel sólo son usados en unas pocas aplicaciones especiales.

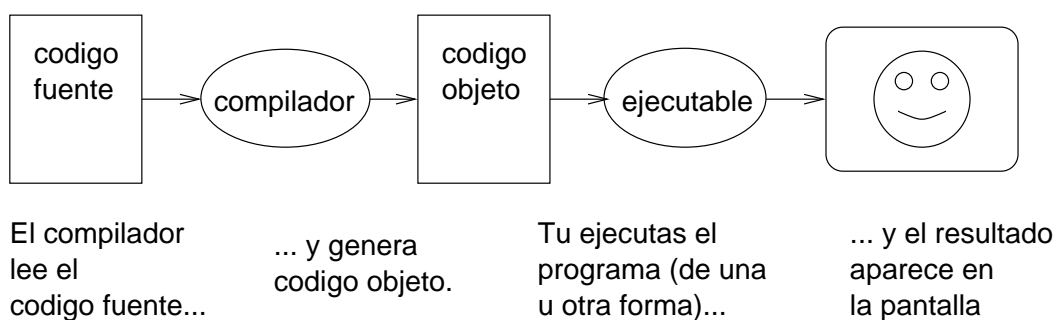
Hay dos maneras de traducir un programa; **interpretar** o **compilar**. Un intérprete es un programa que lee un programa escrito en un lenguaje de alto nivel, y hace lo que dice. En efecto, traduce el programa línea a línea, alternando la lectura de las líneas y la ejecución de sus instrucciones.



Un compilador es un programa que lee un programa escrito en un lenguaje de alto nivel, y lo traduce todo de una vez, antes de ejecutar cualquiera de sus instrucciones. A menudo compilas el programa en un paso separado, y posteriormente ejecutas el código compilado. En este caso, el programa escrito en un lenguaje de alto nivel se denomina **código fuente**, y el programa traducido se llama **código objeto** o **programa ejecutable**.

Como ejemplo, supón que escribes un programa en C. Podrías usar un editor de texto para escribir el programa (un editor de texto es un procesador de textos sencillo). Cuando acabas el programa, puedes guardarlo en un fichero llamado `programa.c`, donde “programa” es un nombre arbitrario, y el sufijo `.c` es un convenio que indica que el fichero contiene código fuente escrito en lenguaje C.

Después, dependiendo de cómo sea tu entorno de programación, puedes salir del editor de texto y ejecutar el compilador. El compilador puede leer tu código fuente, traducirlo, y crear un nuevo fichero llamado `program.o` que contiene el código objeto, o `program.exe` que contiene el ejecutable.



## 1.2. ¿Qué es un programa?

Un programa es una secuencia de instrucciones que especifican cómo realizar una computación. La computación puede ser algo matemático, como resolver un sistema de ecuaciones o encontrar las raíces de un polinomio, pero también puede ser computación simbólica, como buscar y reemplazar texto en un documento o (bastante extraño) compilar un programa.

Las instrucciones (o comandos, o sentencias) tienen un aspecto diferente en diferentes lenguajes

de programación, pero hay unas cuantas funciones básicas que aparecen en prácticamente cualquier lenguaje:

**entrada:** Recoger datos del teclado, de un fichero, o de algún otro dispositivo.

**salida:** Visualizar datos en la pantalla o enviar datos a un fichero u otro dispositivo.

**cálculo:** Realizar operaciones matemáticas básicas como sumar y multiplicar.

**comprobación:** Comprobar ciertas condiciones y ejecutar la secuencia apropiada de sentencias.

**repetición:** Realizar alguna acción repetidamente, a menudo con alguna variación.

Lo creas o no, esto es casi todo lo que hay. Todo programa que hayas usado alguna vez, no importa lo complicado que sea, está hecho de funciones que tienen más o menos el aspecto de éstas. Por tanto, una manera de describir la programación es decir que es el proceso de romper una tarea grande y compleja en sub-tareas más y más pequeñas hasta que finalmente todas las sub-tareas son lo bastante sencillas como para ser realizadas con una de esas sencillas funciones.

## 1.3. ¿Qué es depurar?

Programar es un proceso complejo, y al ser realizado por seres humanos, a menudo conlleva errores. Por razones un tanto caprichosas, los errores de programación se llaman **bugs**<sup>1</sup> y el proceso de seguirles la pista y corregirlos se llama **depuración**.

Hay varios tipos diferentes de errores que pueden ocurrir en un programa, y es útil distinguir entre ellos con idea de seguirles la pista más rápidamente.

### 1.3.1. Errores en tiempo de compilación

El compilador sólo puede traducir un programa si el programa es sintácticamente correcto; en caso contrario, la compilación falla y no puedes ejecutar tu programa. La **sintaxis** se refiere a la estructura de tu programa y las reglas acerca de esa estructura.

Por ejemplo, en castellano, una frase debe comenzar con una letra mayúscula y acabar en punto. esta frase contiene un error sintáctico. Y esta otra también

Para la mayoría de los lectores, unos cuantos errores sintácticos no son un problema significativo, gracias a lo cual podemos leer los poemas de e e cummings sin recibir mensajes de error.

Los compiladores no son tan misericordiosos. Si hay un sólo error sintáctico en alguna parte de tu programa, el compilador mostrará un mensaje de error y finalizará, y no serás capaz de ejecutar tu programa.

Para hacer las cosas aún peor, hay más reglas sintácticas en Eiffel de las que hay en castellano, y los mensajes de error que obtienes del compilador a menudo no son muy útiles. Durante las primeras semanas de tu carrera como programador, probablemente perderás un montón de tiempo buscando errores sintácticos. A medida que vayas ganando experiencia, por contra, cometerás menos errores y los encontrarás más rápidamente.

---

<sup>1</sup>Del inglés, “bichos”.

### 1.3.2. Errores en tiempo de ejecución

El segundo tipo de error es un error en tiempo de ejecución, así llamado porque el error no aparece hasta que ejecutas el programa.

La buena noticia es que Eiffel tiende a ser un lenguaje **seguro**, lo que significa que los errores en tiempo de ejecución son infrecuentes, especialmente en el tipo de programas que escribiremos en las siguientes semanas.

A medida que avance el curso, probablemente empieces a ver más errores en tiempo de ejecución, especialmente cuando empecemos a hablar de objetos y referencias (Capítulo 8).

En Eiffel, los errores en tiempo de ejecución se llaman **excepciones**, y en algunos entornos aparecen como ventanas o cajas de diálogo que contienen información sobre lo que ha ocurrido y lo que el programa estaba haciendo cuando ocurrió. Esta información es útil para la depuración.

### 1.3.3. Errores lógicos y semánticos

El tercer tipo de error es el error **lógico** o **semántico**. Si hay un error lógico en tu programa, compilará y se ejecutará con éxito, en el sentido de que el ordenador no generará ningún mensaje de error, pero no hará lo que tiene que hacer. Hará alguna otra cosa. Específicamente, hará lo que le dijiste que hiciera.

El problema es que el programa que escribiste no es el programa que querías escribir. El significado del programa (su semántica) está equivocado. Identificar errores lógicos puede ser truculento, pues requiere que vayas retrocediendo mirando la salida del programa e intentando hacerte una idea de lo que está haciendo.

### 1.3.4. Depuración experimental

Una de las habilidades más importantes que vas a adquirir en este curso es la de depurar. Aunque puede ser frustrante, depurar es una de las partes más intelectualmente ricas, desafiantes e interesantes de la programación.

En cierta forma la depuración es como el trabajo de un detective. Te enfrentas con pistas y debes deducir los procesos y sucesos que provocaron los resultados que observas.

Depurar además se parece a una ciencia experimental. Una vez que tienes una idea acerca de lo que va mal, modificas tu programa y vuelves a intentarlo. Si tu hipótesis fue correcta, podrás predecir el resultado de la modificación, y te habrás acercado un paso más hacia un programa que funciona. Si tu hipótesis fue errónea, tendrás que volver con una nueva. Como Sherlock Holmes apuntó, “Cuando has eliminado lo imposible, lo que queda, aunque improbable, debe ser la verdad” (*The Sign of Four*, de A. Conan Doyle).

Para algunas personas, programar y depurar son la misma cosa. Es decir, programar es el proceso de depurar un programa gradualmente hasta que haga lo que quieres. La idea es que debes siempre comenzar con un programa que funcione y que haga *algo*, y hacer pequeñas modificaciones, depurándolas por el camino, hasta que tengas un programa correcto.

Por ejemplo, Linux es un sistema operativo que contiene miles de líneas de código, pero comenzó como un simple programa que Linus Torvalds usó para explorar el procesador Intel 80386. Según Larry Greenfield, “Uno de los primeros proyectos de Linus fue un programa que pudiera conmutar entre imprimir AAAA y BBBB. Eso después evolucionó a Linux” (de *The Linux Users’ Guide* Beta Versión 1).

En posteriores capítulos haré más sugerencias sobre depuración y otras prácticas de programación.

## 1.4. Lenguajes formales y naturales

Los **lenguajes naturales** son los lenguajes<sup>2</sup> que habla la gente, como el castellano, el inglés y el francés. No fueron diseñados por la gente (aunque la gente intente imponer algo de orden sobre ellos); evolucionan de forma natural.

Los **lenguajes formales** son lenguajes diseñados por las personas para aplicaciones específicas. Por ejemplo, la notación que los matemáticos utilizan es un lenguaje formal particularmente bueno para representar relaciones entre números y símbolos. Los químicos usan un lenguaje formal para representar la estructura química de las moléculas. Y más importante:

**Los lenguajes de programación son lenguajes formales diseñados para expresar computaciones.**

Como mencioné antes, los lenguajes formales tienden a tener estrictas reglas de sintaxis. Por ejemplo,  $3 + 3 = 6$  es una expresión matemática correcta, pero  $3 = +6\$$  no lo es. Igualmente,  $H_2O$  es un nombre químico sintácticamente correcto, pero  $_2Zz$  no.

Las reglas sintácticas cubren dos aspectos, correspondientes a los *tokens* y a la estructura. Los *tokens* son los elementos básicos del lenguaje, como palabras y números y elementos químicos. Uno de los problemas con  $3=+6\$$  es que  $\$$  no es un *token* permitido en Matemáticas (al menos por lo que yo sé). De igual manera,  $_2Zz$  no es correcto porque no existe ningún elemento con la abreviatura  $Zz$ .

El segundo tipo de error sintáctico se corresponde con la estructura de una frase; es decir, la manera en la que se organizan los *tokens*. La sentencia  $3=+6\$$  es estructuralmente ilegal, porque no puedes tener un signo “más” inmediatamente después de un signo “igual”. Igualmente, las fórmulas moleculares deben tener subíndices después del nombre del elemento, y no antes.

Cuando lees una frase en castellano o en un lenguaje formal, debes darte cuenta de la estructura que posee esa frase (aunque en un lenguaje natural lo haces inconscientemente). Ese proceso se denomina **análisis sintáctico**.

Por ejemplo, cuando oyes la frase “El otro zapato cayó”, entiendes que “el otro zapato” es el sujeto y que “cayó” es el verbo. Una vez que has analizado sintácticamente una frase, puedes entender lo que significa, es decir, la semántica de la frase. Asumiendo que sabes lo que es un zapato, y lo que significa caer, entenderás la implicación general de esta frase.

Aunque los lenguajes formales y naturales tienen muchas características en común —*tokens*, estructura, sintaxis y semántica— hay muchas diferencias.

**ambigüedad:** Los lenguajes naturales están llenos de ambigüedades, que son tratadas por las personas usando pistas contextuales y otra información. Los lenguajes formales se diseñan para ser casi o completamente no ambiguos, lo que significa que toda frase tiene exactamente un significado, independientemente del contexto.

**redundancia:** Con idea de tratar la ambigüedad y reducir los equívocos, los lenguajes naturales emplean gran cantidad de redundancia. Como resultado, a menudo son locuaces. Los lenguajes formales son menos redundantes y más concisos.

**literalidad:** Los lenguajes naturales están llenos de modismos y metáforas. Si yo digo “El otro zapato cayó”, es probable que no exista ningún zapato y que nada caiga. Los lenguajes formales significan exactamente lo que dicen.

A la gente que crece hablando un lenguaje natural (todos y cada uno de nosotros) a menudo les cuesta un tiempo adaptarse a los lenguajes formales. En cierta forma la diferencia entre un lenguaje formal y uno natural se parece a la diferencia entre la poesía y la prosa:

<sup>2</sup>Sería más correcto decir “lenguas” o “idiomas”.

**Poesía:** Se usan las palabras por su sonido así como por su significado, y el poema al completo crea un efecto o respuesta emocional. La ambigüedad no sólo es común sino a menudo deliberada.

**Prosa:** El significado literal de las palabras es más importante y la estructura contribuye a una mayor comprensión. La prosa es más dócil para el análisis que la poesía, pero aún así con frecuencia sigue siendo ambigua.

**Programas:** El significado de un programa de ordenador es literal y no ambiguo, y puede ser comprendido completamente mediante el análisis de los *tokens* y la estructura.

He aquí algunas sugerencias para leer programas (y otros lenguajes formales). Primero, recuerda que los lenguajes formales son mucho más densos que los lenguajes naturales, así que te llevará más tiempo leerlos. Además, la estructura es muy importante, por lo que usualmente no es una buena idea leerlo de arriba abajo, de izquierda a derecha. En cambio, aprende a analizar el programa en tu cabeza, identificando los *tokens* e interpretando la estructura. Finalmente, recuerda que los detalles importan. Pequeñas cosas como errores de ortografía y puntuación incorrecta, las cuales puedes obviar en los lenguajes naturales, pueden suponer una gran diferencia en un lenguaje formal.

## 1.5. El primer programa

Tradicionalmente, el primer programa que la gente escribe en un nuevo lenguaje se llama “¡Hola, mundo!” porque todo lo que hace es escribir las palabras “¡Hola, mundo!”. En Eiffel, este programa tiene el siguiente aspecto:

```
class HOLA

creation make

feature
  make is
    -- make: genera algo de salida simple
    do
      print("¡Hola, mundo!")
    end
end
```

Algunas personas juzgan la calidad de un lenguaje de programación por la simplicidad del programa “¡Hola, mundo!”. Según este estándar, Eiffel no lo hace muy bien. Incluso el programa más simple contiene un número de características que son difíciles de explicar a los programadores novatos. Vamos a ignorar muchas de ellas por ahora, pero explicaré algunas.

Todos los programas están compuestos de *definiciones de clase*, que tienen la siguiente forma:

```
class NOMBRE_CLASE

creation make

feature
  make is
    do
      SENTENCIAS
    end
end
```

Aquí `NOMBRE.CLASE` indica un nombre arbitrario que puedes asignar. El nombre de la clase del ejemplo es `HOLA`.

En la segunda línea, tenemos las palabras **creation** `make`, que indican que `make` (que se define más abajo) será el lugar donde comenzará la ejecución dentro de la clase. Cuando el programa se ejecuta, comienza ejecutando la primera sentencia contenida en `make`, y continúa, en orden, hasta que alcanza su última sentencia, tras lo cual finaliza. La definición de `make` comienza tras la palabra **feature**.

No existe límite para el número de sentencias que pueden aparecer en `make`, pero el ejemplo contiene sólo una. Es una sentencia de **impresión**, que lo que hace es visualizar un mensaje en la pantalla. Es un poco confuso el que “impresión” a veces signifique “visualizar algo en la pantalla”, y otras veces signifique “enviar algo a la impresora”. En este libro no hablaré mucho acerca de mandar cosas a la impresora; haremos todas nuestras impresiones sobre la pantalla.

El comando que imprime cosas en la pantalla es **print**, y lo que hay entre los paréntesis es lo que se va a imprimir.

Hay algunas otras cosas que habrás observado sobre la sintaxis de este programa. Primero, Eiffel usa la palabra **end** para indicar el final de algo. La definición de `make` comienza con la palabra `do` y acaba con la primera aparición de la palabra **end**. La definición de la clase completa comienza con **class** `HOLA` y acaba en la última aparición de **end** (en la última línea del programa). Observa que la definición de la clase incluye dentro la definición de `make`.

Además, observa que la línea que hay a continuación de `make` **is** comienza con `--`. Esto indica que esta línea contiene un **comentario**, que es un poco de texto en castellano que puedes colocar en medio de un programa, normalmente para explicar lo que hace el programa. Cuando el compilador encuentra el `--`, ignora todo lo que haya desde ahí hasta el final de la línea.

Para poder compilar tus programas, tienes que indicar expresamente al compilador que la ejecución comenzará en la primera sentencia de `make`, el cual está definido a su vez en la clase `HOLA`. Por tanto `make` será lo que se llama el **método raíz**, y `HOLA` será lo que se llama la **clase raíz**. La ejecución de un programa Eiffel comienza siempre en el método raíz de la clase raíz. En cada compilador y en cada entorno de desarrollo Eiffel debe haber alguna manera de indicar cuáles son el método y la clase raíz de nuestro programa. Consulta para ello la documentación del compilador o el entorno que utilices.

En tu primera práctica compila y ejecuta este programa, y además modifícalo de varias formas con idea de ver qué son las reglas sintácticas, y ver qué errores genera el compilador cuando violas una de esas reglas. Recuerda que para poder compilar tu programa, debes indicar al compilador que la clase raíz es `HOLA` y que el método raíz es `make`.

## 1.6. Glosario

**resolución de problemas:** El proceso de formular un problema, encontrar una solución, y expresar la solución.

**lenguaje de alto nivel:** Un lenguaje de programación como Eiffel diseñado para ser fácil de leer y escribir para los humanos.

**lenguaje de bajo nivel:** Un lenguaje de programación diseñado para ser fácil de ejecutar por un ordenador. También llamado “lenguaje máquina” o “lenguaje ensamblador”.

**lenguaje formal:** Cualquiera de los lenguajes que la gente diseña para propósitos específicos, como representar ideas matemáticas o programas de ordenador. Todos los lenguajes de programación son lenguajes formales.

**lenguaje natural:** Cualquiera de los lenguajes que la gente habla y que evolucionan de manera natural.



**transportabilidad:** La propiedad que tiene un programa de poder ser ejecutado en más de un tipo de ordenador.

**interpretar:** Ejecutar un programa en un lenguaje de alto nivel traduciéndolo línea a línea.

**compilar:** Traducir un programa en un lenguaje de alto nivel a un lenguaje de bajo nivel, todo a la vez, preparándolo para una ejecución posterior.

**código fuente:** Un programa escrito en un lenguaje de alto nivel, antes de ser compilado.

**código objeto:** La salida del compilador, después de traducir el programa.

**ejecutable:** Otro nombre para el código objeto que está listo para ser ejecutado.

**algoritmo:** Un proceso general para resolver una categoría de problemas.

**bug:** Un error en un programa.

**sintaxis:** La estructura de un programa.

**semántica:** El significado de un programa.

**análisis sintáctico:** Examinar un programa y analizar su estructura sintáctica.

**error sintáctico:** Un error en un programa que lo hace imposible de analizar sintácticamente (y por tanto imposible de compilar).

**excepción:** Un error en un programa que hace que falle en tiempo de ejecución. También llamado error en tiempo de ejecución.

**error lógico:** Un error en un programa que hace que haga algo distinto a lo que el programador pretendía.

**depuración:** El proceso de encontrar y eliminar cualquiera de los tres tipos de errores.

## Capítulo 2

# Variables y tipos

### 2.1. Más impresión

Como mencioné en el último capítulo, puedes colocar todas las sentencias que quieras en **make**. Por ejemplo, para imprimir más de una línea.

```
class HOLA

  creation make

  feature
    make is
      -- make: genera algo de salida simple
    do
      print("¡Hola, mundo!%N") -- imprime una línea
      print("¿Cómo estás?")    -- imprime otra
    end
  end

end
```

Además, como puedes ver, es legal colocar comentarios al final de una línea, igual que en una línea por sí mismos.

Las frases que aparecen entre comillas son **cadenas**, porque están formadas por una secuencia (cadena) de letras. En realidad, las cadenas pueden contener cualquier combinación de letras, números, símbolos de puntuación y otros caracteres especiales.

**print** imprime una cadena, pero al hacerlo, no mueve el cursor a la siguiente línea de la pantalla. Si queremos que cada cadena aparezca en una línea diferente, debemos añadir a la primera cadena un carácter especial llamado el carácter de **nueva línea**. En Eiffel, el carácter de nueva línea se representa por los caracteres **%N** dentro de una cadena, como puede verse al final de la cadena “¡Hola, mundo!”. Cuando Eiffel imprime una cadena y encuentra la combinación **%N**, hace que el cursor salte a la línea siguiente, lo que en nuestro caso provoca que la siguiente cadena aparezca en la línea siguiente de la pantalla.

Si quisiéramos mostrar la salida de múltiples sentencias **print** en una misma línea, podríamos hacerlo con el comando **print** sin hacer uso del carácter nueva línea:

```
class HOLA

  creation make
```

```

feature
  make is
    -- make: genera algo de salida simple
  do
    print("¡Adiós, ")
    print("mundo cruel!")
  end
end

```

En este caso la salida aparece en una sola línea como `¡Adiós, mundo cruel!`. Observa que hay un espacio entre la coma y las segundas comillas. Este espacio aparece en la salida, por lo que afecta al comportamiento del programa.

Los espacios en blanco que aparecen fuera de las comillas generalmente no afectan al comportamiento del programa. Por ejemplo, podría haber escrito:

```

class HOLA

creation make

feature
make is
do
print("¡Adiós, ")
print("mundo cruel!")
end
end

```

Este programa compila y se ejecuta tan bien como el original. Las divisiones al final de las líneas (nuevas líneas) no afectan tampoco al comportamiento del programa, por lo que podría haber escrito:

```

class HOLA creation make feature make is do
print("¡Adiós, ") print("mundo cruel!") end end

```

Esto también funcionaría, aunque probablemente te hayas dado cuenta de que el programa se hace más y más complicado de leer. La separación de líneas y los espacios en blanco son útiles para organizar visualmente tu programa, haciéndolo más fácil de leer y de localizar errores sintácticos.

## 2.2. Variables

Una de las características más poderosas de un lenguaje de programación es su capacidad para manipular **variables**. Una variable es un lugar, etiquetado con un nombre, en el que se almacena un **valor**. Los valores son cosas que pueden ser visualizadas y almacenadas y (como veremos luego) se puede operar con ellas. Las cadenas que hemos visualizado (`"¡Hola, mundo!"`, `"¡Adiós, "`, etc.) son valores.

Para almacenar un valor, debes crear una variable. Como los valores que queremos almacenar son cadenas, indicaremos que la nueva variable es una cadena:

```

pepe: STRING

```

Esta sentencia es una **declaración**, porque declara que la variable llamada **pepe** tiene el tipo **STRING** (cadena). Cada variable tiene un tipo que determina qué tipo de valores puede almacenar. Por ejemplo, el tipo **INTEGER** (entero) puede almacenar números enteros, y probablemente no te sorprenderá el que el tipo **STRING** pueda almacenar cadenas.

Para crear una variable entera, la sintaxis es **juan: INTEGER**, donde **juan** es el nombre arbitrario que le das a la variable. En general querrás nombrar a las variables de manera que sus nombres indiquen lo que planeas hacer con esa variable. Por ejemplo, si ves estas declaraciones de variable:

```
nombre: STRING
apellidos: STRING
horas, minutos: INTEGER
```

probablemente podrás hacerte una suposición muy acertada acerca de los valores que se van a almacenar en ellas. Este ejemplo muestra además la sintaxis para declarar varias variables con el mismo tipo: **horas** y **minutos** son enteras (tipo **INTEGER**).

En Eiffel, las variables se declaran en una sección especial entre la línea **make is** (incluyendo la posible línea de comentario que empieza por **--**) y la que contiene la palabra **do**. Entre esas dos líneas se incluye la palabra **local** seguida de las declaraciones de variable. Por ejemplo:

```
class HOLA

creation make

feature
  make is
    -- make: ejemplo de declaración de variables
  local
    nombre: STRING
    apellidos: STRING
    horas, minutos: INTEGER
  do
    -- aquí vendrían las sentencias
  end
end
```

Por tanto, no pueden aparecer declaraciones de variables entre las palabras **do** y **end**, como si fueran sentencias ordinarias.

## 2.3. Asignación

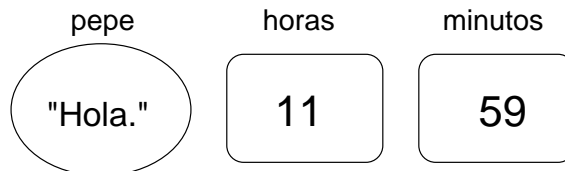
Ahora que hemos creado algunas variables, nos gustaría almacenar valores en ellas. Lo haremos con una **sentencia de asignación**.

```
pepe := "Hola."      -- da a 'pepe' el valor "Hola."
horas := 11           -- asigna el valor 11 a 'horas'
minutos := 59         -- pone 'minutos' a 59
```

Este ejemplo muestra tres asignaciones, y los comentarios reflejan tres formas diferentes que la gente usa a veces al hablar de sentencias de asignación. El vocabulario puede ser confuso aquí, pero la idea está clara:

- Cuando declaras una variable, creas un lugar de almacenamiento con un nombre.
- Cuando realizas una asignación a una variable, le das un valor.

Una manera habitual de representar variables en el papel es dibujar una caja con el nombre de la variable fuera, y el valor de la variable dentro. Esta figura muestra el efecto de las tres sentencias de asignación:



Llamaremos a este tipo de diagramas **diagramas de estado**. Para estos diagramas usaré formas diferentes para representar tipos diferentes de variable. Estas formas deberían ayudarte a recordar que una de las reglas en Eiffel es que una variable debe tener el mismo tipo que el valor que le asignas. No puedes almacenar una cadena en `minutos`, o un entero en `pepe`. Sería como meter una clavija cuadrada en un agujero redondo.

Una fuente de confusión es que algunas cadenas *parecen* enteros, pero no lo son. Por ejemplo, `pepe` puede contener la cadena "123", que está hecha de los caracteres 1, 2 y 3, pero no es lo mismo que el *número* 123.

```

pepe := "123";    -- esto está permitido
pepe := 123       -- esto no está permitido
  
```

## 2.4. Imprimir variables

Puedes imprimir el valor de una variable usando las mismas acciones que usamos para imprimir cadenas.

```

class HOLA

creation make

feature
  make is
    local
      primera_linea: STRING
    do
      primera_linea := "¡Hola de nuevo!"
      print(primera_linea)
    end
end
  
```

Este programa crea una variable llamada `primera_linea`, le asigna el valor "¡Hola de nuevo!" y después imprime ese valor. Cuando decimos "imprimir una variable", nos referimos a imprimir el *valor* de esa variable. Para imprimir el *nombre* de una variable, debes ponerlo entre comillas. Por ejemplo: `print("primera_linea")`.

Si quieres hacer algo más curioso, puedes escribir:

```

class HOLA

creation make

feature
  make is
    do
      primera_linea := "¡Hola de nuevo!"
      print("El valor de primera_linea es ")
      print(primera_linea)
    end
end
end

```

La salida de este programa es

```

El valor de primera_linea es ¡Hola de nuevo!

```

Me gustaría recalcar el hecho de que la sintaxis para imprimir una variable es la misma con independencia del tipo de la variable.

```

class HORA

creation make

feature
  make is
    local
      horas, minutos: INTEGER
    do
      horas := 11
      minutos := 59
      print("La hora actual es ")
      print(horas)
      print(":")
      print(minutos)
      print(".")
    end
end
end

```

La salida de este programa es `La hora actual es 11:59`.

NOTA: Te recuerdo que, si en algún momento quieres hacer que el cursor de la pantalla salte a la línea siguiente, para que el siguiente valor aparezca en una nueva línea en lugar de en la línea actual, debes imprimir el carácter de nueva línea, representado por `%N`. Esto lo puedes hacer en una instrucción independiente, como `print("%N")`, o adjuntando la secuencia `%N` al final de una cadena, como en `print("La hora actual es %N")`. En este último caso, tras mostrar la cadena `La hora actual es`, el cursor saltará a la línea siguiente, y los nuevos valores que mandemos imprimir con `print` aparecerán en una nueva línea.

Si quieres hacer tus programas aún más legibles, en lugar de usar `print("%N")` para pasar a la siguiente línea, puedes usar `io.put_new_line`, puesto que ambas acciones son equivalentes.

## 2.5. Palabras clave

Unas cuantas secciones atrás, dije que puedes darle cualquier nombre que quieras a tus variables, pero eso no es totalmente cierto. Existen ciertas palabras que están reservadas en Eiffel porque son usadas por el compilador para analizar sintácticamente la estructura de tu programa, y si las usas como nombres de variable, se confundirá. Estas palabras, llamadas **palabras clave**, incluyen `class`, `creation`, `feature`, `local`, `do`, `is`, `end` y muchas más.

En lugar de tener que memorizar la lista de palabras clave, te sugiero que aproveches una característica proporcionada por muchos entornos de desarrollo para Eiffel: el resaltado de código. Mientras escribes, diferentes partes de tu programa deberían aparecer en colores diferentes. Por ejemplo, las palabras clave podrían aparecer en azul, las cadenas en rojo, y el resto del código en negro. Si tecleas el nombre de una variable y aparece en azul, ¡cuidado! Obtendrás un extraño comportamiento del compilador.

## 2.6. Operadores

Los **operadores** son símbolos especiales que se utilizan para representar cálculos simples como la suma o la multiplicación. Muchos de los operadores en Eiffel hacen exactamente lo que podrías esperar de ellos, porque son símbolos matemáticos comunes. Por ejemplo, el operador para sumar dos enteros es `+`.

Las siguientes son todas expresiones legales de Eiffel cuyo significado es más o menos obvio:

```
1 + 1      horas - 1      horas * 60 + minutos      minutos / 60
```

Las expresiones pueden contener nombres de variable y números. En cada caso el nombre de una variable se sustituye por su valor antes de que se realice el cálculo.

La suma, la resta y la multiplicación hacen lo que esperas, pero debes tener cuidado con la división. Por ejemplo, el siguiente programa:

```
class HORAS

creation make

feature
  make is
    local
      horas, minutos: INTEGER
    do
      horas := 11
      minutos := 59
      print("Número de minutos desde medianoche: ")
      print(horas * 60 + minutos)
      io.put_new_line    -- igual que print("%N")
      print("Fracción de hora que ha transcurrido: ")
      print(minutos / 60)
    end
end
```

generaría la siguiente salida:

```
Número de minutos desde medianoche: 719
Fracción de hora que ha transcurrido: 0.983333
```

El resultado es correcto, pero observa la segunda línea. El resultado de dividir `minutos` (que es una variable entera) entre 60 (que es una cantidad también entera) es un número no entero, ya que contiene decimales. En Eiffel (y en la mayoría de los lenguajes de programación), un número que contiene decimales se denomina **número real**. La división es la única operación que proporciona siempre como resultado un número real, independientemente del tipo de los números que estemos dividiendo.

## 2.7. Orden de las operaciones

Cuando en una expresión aparece más de un operador, el orden de evaluación depende de las reglas de **prioridad**. Una exposición completa del tema de la prioridad puede llegar a ser complejo, pero para empezar aquí tienes algunos puntos:

- La multiplicación y la división tienen prioridad sobre (se hacen antes que) la suma y la resta. Así,  $2*3-1$  da 5, no 4, y  $2/3-1$  da  $-0,333333$ , no 1.
- Si los operadores tienen la misma prioridad, entonces son evaluados de izquierda a derecha. Así en la expresión `minutos-100+60`, la resta aparece primero, dando 19. Si las operaciones se hicieran de derecha a izquierda, el resultado hubiera sido `minutos-160` que es  $-101$ , lo que es incorrecto.
- Siempre que quieras saltarte las reglas de prioridad (o si no estás seguro de cuáles son) puedes usar paréntesis. Las expresiones entre paréntesis se evalúan primero, así  $2*(3-1)$  es 4. Además, puedes usar los paréntesis para hacer que una expresión sea más fácil de leer, como en  $(minutos*100)/60$ , aunque eso no cambie el resultado.

## 2.8. Operadores sobre cadenas (STRINGS)

En general, no puedes realizar operaciones matemáticas sobre cadenas, incluso si las cadenas tienen el aspecto de números. Lo siguiente es incorrecto (si sabemos que `pepe` es de tipo **STRING**).

```
pepe - 1          "Hola" / 123      pepe * "Hola"
```

Por cierto, ¿puedes decirme, mirando estas expresiones, si `pepe` es un entero o es una cadena? No. La única manera de determinar el tipo de una variable es mirar el sitio donde fue declarada.

Curiosamente, el operador `+` funciona con cadenas, aunque no hace exactamente lo que te podrías esperar. Para las cadenas, el operador `+` representa la **concatenación**, que significa juntar los dos operandos poniendo uno a continuación del otro. Así `"Hola, " + "mundo."` produce la cadena `"Hola, mundo."` y `pepe + "ísimo"` añade el sufijo *-ísimo* al final de lo que sea `pepe`, lo que a menudo es útil para dar nombre a nuevas formas de intolerancia.

## 2.9. Composición

Hace tiempo vimos los elementos de un lenguaje de programación —variables, expresiones y sentencias— por separado, sin hablar nada acerca de cómo combinarlos.

Una de las características más útiles de los lenguajes de programación es su habilidad de tomar pequeños bloques de construcción y **componerlos**. Por ejemplo, sabemos cómo multiplicar enteros y sabemos cómo imprimirlos; pues entonces podemos hacer las dos cosas al mismo tiempo:



```
print(17 * 3)
```

Realmente, no deberíamos decir “al mismo tiempo”, ya que en realidad la multiplicación se hace antes que la impresión, pero la idea es que cualquier expresión que incluya números, cadenas y variables, puede ser usada dentro de una sentencia de impresión. Ya hemos visto un ejemplo:

```
print(horas * 60 + minutos)
```

Pero también puedes colocar expresiones arbitrarias en la parte derecha de una sentencia de asignación:

```
class RAIZ
  creation make
  feature
    make is
      local
        porcentaje: INTEGER
      do
        porcentaje := (minutos * 100) / 60
      end
    end
end
```

Esta habilidad no parece muy impresionante ahora, pero veremos otros ejemplos donde la composición hace posible expresar cálculos complejos limpia y concisamente.

NOTA: Hay límites sobre dónde puedes usar ciertas expresiones; como ejemplo notable, la parte izquierda de una sentencia de asignación debe ser un *nombre* de variable, no una expresión. Esto es por que la parte izquierda indica el lugar de almacenamiento a donde irá el resultado. Las expresiones no representan lugares de almacenamiento: sólo valores. Así, lo siguiente es incorrecto: `minutos+1 := horas`.

## 2.10. Glosario

**variable:** Un lugar de almacenamiento de valores etiquetado con un nombre. Todas las variables tienen un tipo, el cual se declara cuando se crea la variable.

**valor:** Un número o cadena (u otras cosas que se verán después) que puede ser almacenado en una variable. Cada valor pertenece a un tipo.

**tipo:** Un conjunto de valores. El tipo de una variable determina qué valores puede almacenar. Hasta ahora, los tipos que hemos visto han sido enteros (`INTEGER` en Eiffel) y cadenas (`STRING` en Eiffel).

**palabra clave:** Una palabra reservada utilizada por el compilador para analizar programas sintácticamente. No puedes usar esas palabras (`class`, `creation`, `feature`...) como nombres de variable.

**sentencia:** Una línea de código que representa una acción. Hasta ahora, las sentencias que hemos visto han sido declaraciones (en una sección aparte etiquetada con `local`), asignaciones y sentencias de impresión.

**declaración:** Una sentencia que crea una nueva variable y determina su tipo.

**asignación:** Una sentencia que asigna un valor a una variable.

**expresión:** Una combinación de variables, operadores y valores que representan un único valor resultado. Las expresiones también tienen tipos, determinados por sus operadores y operandos.

**operador:** Un símbolo especial que representa un cálculo sencillo, como sumar, restar o concatenar cadenas.

**operando:** Uno de los valores sobre los que actúa un operador.

**prioridad:** El orden en el que se evalúan las operaciones.

**concatenar:** Unir dos operandos uno a continuación del otro.

**composición:** La capacidad de combinar expresiones y sentencias simples en expresiones y sentencias compuestas con la finalidad de representar cálculos complejos de manera concisa.

## Capítulo 3

# Métodos

### 3.1. Reales

En el último capítulo tuvimos algunos problemas con números que no eran enteros. Arreglamos el problema midiendo porcentajes en lugar de fracciones, pero la solución más general es usar números reales, los cuales puede representar con decimales así como enteros. En Eiffel, el tipo real se llama `DOUBLE`.

Puedes crear variables reales y asignarles valores usando la misma sintaxis que usamos para los otros tipos. Por ejemplo:

```
local
  pi: DOUBLE
do
  pi := 3.14159
end
```

Evidentemente, esta porción de código no es suficiente para formar un programa completo que pueda compilarse, porque no hay ninguna definición de clase, ni `creation`, ni `feature`, ni nada. Pero sí podría ser la definición de `make`, por ejemplo. Así que, a partir de ahora, y para abreviar, cuando te muestre un trozo de código como el anterior, debes entender que en realidad quería decir algo así:

```
class RAIZ

  creation make

  feature
    make is
      local
        pi: DOUBLE
      do
        pi := 3.14159
      end
    end
end
```

Esto sí es un programa que puede compilarse y ejecutarse.

Aunque los números reales son útiles, a menudo son fuente de confusión porque parece haber un solapamiento entre los números enteros y los reales. Por ejemplo, si tienes el valor 1, ¿es un entero, un real, o ambas cosas?

Estrictamente hablando, Eiffel distingue el valor entero 1 del valor real 1.0, incluso aunque parezcan ser el mismo número. Pertenecen a tipos diferentes, y estrictamente hablando, no se te está permitido hacer asignaciones entre tipos. Por ejemplo, lo siguiente es incorrecto<sup>1</sup>:

```
local
  x: INTEGER
do
  x := 1.1
end
```

porque la variable a la izquierda de la asignación es un `INTEGER` y el valor a la derecha es un `DOUBLE`. Pero es fácil olvidar esa regla, especialmente porque existen lugares en los que Eiffel convierte automáticamente de un tipo a otro. Por ejemplo:

```
local
  y: DOUBLE
do
  y := 1
end
```

técnicamente no debería ser legal, pero Eiffel lo permite convirtiendo el `INTEGER` a `DOUBLE` automáticamente.

Todas las operaciones que hemos visto hasta ahora —suma, resta, multiplicación y división— también funcionan sobre valores reales, aunque puede que te interese saber que el mecanismo subyacente es totalmente diferente. De hecho, muchos procesadores disponen de hardware especial sólo para realizar operaciones con números reales.

### 3.2. Convertir un `DOUBLE` en un `INTEGER`

Como ya mencioné, Eiffel convierte `INTEGERs` a `DOUBLEs` automáticamente si es necesario, porque no se pierde información en la traducción. En cambio, pasar de un `DOUBLE` a un `INTEGER` requiere redondear. Eiffel no realiza esta operación automáticamente, con idea de asegurarse de que tú, como programador, estás avisado de la pérdida de la parte decimal del número.

La manera más sencilla de convertir un número real en un entero es pedirle al número que nos diga qué valor tiene si lo redondeamos. La sintaxis para ello es usar el sufijo `.rounded` a continuación del número real que se quiere redondear. Por ejemplo:

```
class RAIZ

creation make

feature
  make is
    local
      x: DOUBLE
      y: INTEGER
```

---

<sup>1</sup>Independientemente de que no sea un programa completo; recuerda lo que te dije antes.

```

do
    x := 4.76
    y := x.rounded
    print(y)
end
end

```

El resultado de este programa será 5, puesto que *y* valdrá lo que valía *x*, pero redondeado al entero más próximo.

Otras maneras de convertir un número real en un entero es obtener el número entero por defecto o el número entero por exceso. Por ejemplo:

```

class RAIZ

creation make

feature
    make is
        local
            x: DOUBLE
            y: INTEGER
        do
            x := 4.76
            y := x.floor      -- toma el entero por defecto: 4
            print(y)
            x := 6.28
            y := x.ceiling    -- toma el entero por exceso: 7
            print(y)
        end
    end
end

```

El resultado sería

```

4
7

```

El sufijo `.floor` le pide a *x* (el valor que está justo antes del punto), que calcule cuánto vale su parte entera, sin tener en cuenta redondeos. Ese valor es asignado luego a la variable entera *y*. El sufijo `.ceiling` le pide que calcule el siguiente entero que tenga más próximo. Como se puede apreciar, tomar el entero por defecto o el entero por exceso no es lo mismo que redondear.

Como curiosidad, no hace falta guardar el número real en una variable para luego usar los sufijos `.rounded`, `.floor` y `.ceiling`. Se pueden usar esos sufijos directamente sobre un número real, pero en este caso es obligatorio que el número se encierre entre paréntesis:

```

class RAIZ

creation make

feature
    make is
        local
            y: INTEGER

```

```

do
  y := (2.85).rounded      -- redondea a 3
  print(y)
  y := (5.317).ceiling     -- resulta 6
  print(y)
  y := (9.999).floor       -- da 9
  print(y)
end
end

```

Y el resultado, como cabría esperar, es

```

3
6
9

```

Por tanto, no hemos tenido que declarar una variable de tipo `DOUBLE` para guardar los valores reales en ella.

Los sufijos `.rounded`, `.ceiling` y `.floor`, entre otros muchos, son casos particulares de **métodos**. Se puede considerar que los métodos son *peticiones* que se le hacen al valor situado antes del nombre del método (justo antes del punto “.”). Por ejemplo, cuando escribimos `(2.85).rounded`, podemos pensar que le estamos pidiendo al número 2.85 que calcule qué valor tendría si se redondeara al entero más cercano. En este sentido, es como si considerásemos que el número 2.85 es lo bastante inteligente como para saber calcular su valor redondeado. Esta manera de pensar, en la que solicitamos que ciertos objetos nos hagan cosas, es clave en la programación con Eiffel y, en general, con cualquier lenguaje orientado a objetos.

### 3.3. Métodos matemáticos

En Matemáticas, probablemente hayas visto funciones como `cos` y `log`, y aprendido a evaluar expresiones como  $\cos(\pi/2)$  y  $\log(1/x)$ . Primero, evalúas la expresión entre paréntesis, lo que se llama el **argumento** de la función. Por ejemplo,  $\pi/2$  es aproximadamente 1.571, y  $1/x$  es 0.1 (suponiendo que  $x$  es 10).

Después puedes evaluar la función misma, mirando en una tabla o realizando varios cálculos. El `cos` de 1.571 es 0, y el `log` de 0.1 es -1 (suponiendo que `log` indique el logaritmo en base 10).

Este proceso puede ser aplicados repetidamente para evaluar expresiones más complicadas como  $\log(1/\cos(\pi/2))$ . Primero evaluamos el argumento de la función más interna, luego evaluamos la función, y así.

Eiffel proporciona un conjunto de funciones internas que incluyen muchas de las operaciones matemáticas que puedas pensar. Estas funciones también son métodos. La mayoría de los métodos matemáticos operan sobre valores `DOUBLEs`.

Los métodos matemáticos se utilizan usando una sintaxis similar a la que hemos visto para el redondeo de números: colocar un sufijo a continuación del número:

```

class RAIZ

creation make

feature
  make is

```

```

    local
      raiz, angulo, ancho: DOUBLE
    do
      raiz := (17.0).sqrt      -- raíz cuadrada
      angulo := 1.5
      ancho := angulo.cos      -- coseno de 'angulo'
    end
  end
end

```

El primer ejemplo asigna a **raiz** la raíz cuadrada de 17. El segundo ejemplo calcula el coseno de 1.5, que será el valor de la variable **angulo**. Eiffel asume que los valores que puedas usar en **cos** y las demás funciones trigonométricas (**tan**, **sec**) están en *radianes*. Para transformar grados en radianes, puedes dividir por 360 y multiplicar por  $2\pi$ . Afortunadamente, Eiffel proporciona “de fábrica” un valor para  $\pi$ :

```

class RAIZ

creation make

feature
  make is
    local
      grados, angulo: DOUBLE
    do
      grados := 90
      angulo := grados * 2 * Pi / 360.0
    end
  end
end

```

### 3.4. Composición

Al igual que las funciones matemáticas, los métodos de Eiffel se pueden **componer**, lo que significa que puedes usar una expresión como parte de otra. Por ejemplo, puedes usar una expresión como parte de la llamada a un método:

```

local
  x, angulo: DOUBLE
do
  angulo := 3
  x := (angulo + Pi / 2).cos
end

```

La última sentencia toma el valor **Pi**, lo divide por dos y suma el resultado al valor de la variable **angulo**. Después, al número resultante se le pide que calcule su coseno usando el sufijo **.cos**.

También puedes tomar el resultado de un método y usarlo en otro:

```

local
  x: DOUBLE
do
  x := (10.0).log.exp
end

```

En Eiffel, el método `log` siempre usa la base  $e$ , por lo que esta sentencia busca el logaritmo en base  $e$  de 10 y luego eleva  $e$  a esa potencia. El resultado al final se asigna `x`; espero que sepas cuáles es.

### 3.5. Añadir nuevos métodos

Hasta ahora sólo hemos usado los métodos que vienen internamente en Eiffel, pero también es posible añadir nuevos métodos. En realidad, ya hemos visto una definición de método: `make`. El método llamado `make` dijimos que es especial porque hacemos que la ejecución del programa comience por él, pero la sintaxis para `make` es la misma que para cualquier otra definición de método:

```
NOMBRE (LISTA DE PARÁMETROS) is
do
    SENTENCIAS
end
```

Y por supuesto esa definición de método debe ir dentro de una definición de clase, así que sería más correcto escribirlo así:

```
class MI_CLASE
-- ...
feature
-- ...
    NOMBRE (LISTA DE PARÁMETROS) is
    do
        SENTENCIAS
    end
end
```

Los puntos suspensivos “...” indican que pueden existir otros elementos sintácticos allí donde aparecen. Por ejemplo, después de `class MI_CLASE` puede haber un `creation make`, y después de `feature` podría haber una definición de algún otro método, como `make`. Simplemente, con esos “...” estoy indicando que los demás elementos de la clase `MI_CLASE` no son interesantes ahora, y así puedo abreviar algo durante la explicación. Por último, recuerda que si te muestro la definición de un método y esa definición no la incluyo dentro de ninguna clase, a menos que diga lo contrario, esa definición debe ir dentro de la clase raíz.

Puedes darle al método el nombre que quieras, a excepción de que no sea una palabra clave de Eiffel. La lista de parámetros especifica qué información, si la hay, debes proporcionar para usar (o **invocar**) el nuevo método.

El primer par de métodos que vamos a escribir no tiene parámetros, por lo que la sintaxis se parecerá a esto:

```
nueva_linea is
do
    print("%N")      -- o también io.put_new_line
end
```

Este método se llama `nueva_linea`, y como se observa no posee parámetros, pues no hay nada entre paréntesis entre el nombre y la palabra `is`. Contiene una única sentencia, que como ya sabemos, hace que el cursor de la pantalla salte a la siguiente línea.

En `make` podemos invocar a este nuevo método usando una sintaxis similar a la que usamos para invocar los métodos internos de Eiffel:



```

make is
do
    print("Primera línea.%N")
    nueva_linea
    print("Segunda línea.%N")
end

```

La salida de este programa es

```
Primera línea.
```

```
Segunda línea.
```

Observa el espacio extra entre las dos líneas. ¿Qué pasaría si quisiéramos mas espacio entre las líneas? Podríamos invocar el mismo método varias veces:

```

make is
do
    print("Primera línea.%N")
    nueva_linea
    nueva_linea
    nueva_linea
    print("Segunda línea.%N")
end

```

O podríamos escribir un nuevo método, llamado `tres_lineas`, que escriba tres nuevas líneas:

```

tres_lineas is
do
    nueva_linea nueva_linea nueva_linea
end

make is
do
    print("Primera línea.%N")
    tres_lineas
    print("Segunda línea.%N")
end

```

Podrás observar unas cuantas cosas en este programa:

- Puedes invocar el mismo método repetidamente. De hecho, es bastante común y útil hacerlo.
- Puedes hacer que un método llame a otro método. En este caso, `make` llama a `tres_lineas` y `tres_lineas` llama a `nueva_linea`. De nuevo, esto es común y útil.
- En `tres_lineas` escribí tres sentencias en la misma línea, lo que es sintácticamente legal (recuerda que los espacios y las nuevas líneas no cambian el significado de un programa). Por otra parte, normalmente es una buena idea situar cada sentencia línea en una línea aparte, para hacer tu programa más fácil de leer. A veces romperé esa regla en este libro para ganar espacio.

Hasta ahora, no queda muy claro por qué hemos creado todos estos nuevos métodos. En realidad, existen un montón de razones, pero este ejemplo sólo demuestra dos:

1. Crear un nuevo método te da la oportunidad de dar un nombre a un grupo de sentencias. Los métodos pueden simplificar un programa ocultando un cálculo complejo detrás de una acción simple, y usando palabras en castellano en lugar de código oscuro. ¿Qué queda más claro, `nueva_linea` o `print("%N")`?
2. Crear un método puede hacer que un programa quede más corto eliminando código repetitivo. Por ejemplo ¿cómo dejarías nueve líneas en blanco? Bastaría con llamar a `tres_lineas` tres veces.

### 3.6. Clases y métodos

Juntando todos los fragmentos de código de la sección anterior, la definición completa de la clase quedaría así:

```
class NUEVA_LINEA

  creation make

  feature
    nueva_linea is
      do
        print("%N")
      end

    tres_lineas is
      do
        nueva_linea nueva_linea nueva_linea
      end

  make is
    do
      print("Primera línea.%N")
      tres_lineas
      print("Segunda línea.%N")
    end

end
```

La primera línea indica que esta es la definición de clase para una nueva clase llamada `NUEVA_LINEA`. Una clase es una colección de métodos relacionados. En este caso, la clase `NUEVA_LINEA` contiene tres métodos, llamados `nueva_linea`, `tres_lineas` y `make`. Si quieres compilar y ejecutar este programa, no olvides indicar al compilador que la clase raíz es `NUEVA_LINEA`, y que el método raíz es `make`.

También hemos visto otros métodos como `rounded`, `cos`, `log` y `exp`. Observarás que hay una ligera diferencia entre invocar a estos métodos e invocar a los que hemos creado nosotros: los métodos `rounded`, `cos`, etcétera se usan como sufijos de los valores sobre los que actúan (por ejemplo, `(37.44).rounded`), mientras que `nueva_linea` y `tres_lineas` se invocan simplemente poniendo su nombre, sin puntos ni nada delante.

Por ahora es suficiente con que sigas la siguiente regla: cuando dos métodos estén definidos dentro de la misma clase (como ocurre con `tres_lineas` y `nueva_linea`), y uno de ellos llame al otro, no será necesario poner el nombre del método llamado como sufijo de nada. Sin embargo, cuando un método llame a otro definido fuera de su clase (lo que ocurre con todos los métodos que vienen con Eiffel “de fábrica”, como `rounded` o `log`) deberá usarse el nombre del método invocado

como sufijo de algo. Es por eso por lo que `tres_lineas` llama a `nueva_linea` simplemente poniendo su nombre, mientras que para llamar a `cos` hay que usar su nombre como sufijo detrás del valor al que se le quiere calcular el coseno (como en `angulo.cos`).

Como honrosa excepción a esta regla, `print` es un método predefinido de Eiffel, y sin embargo lo invocamos como si fuera un método definido por nosotros en la clase actual. Considera esto como un caso especial hasta que lleguemos a ver algo de la *herencia*.

### 3.7. Programas con varios métodos

Cuando miras una definición de clase que contiene varios métodos, te tienta leerla de arriba abajo, pero esto suele confundir, porque no es ése el **orden de ejecución** del programa.

Para nosotros, la ejecución siempre comenzará en la primera sentencia del método `make` — porque le decimos al compilador que es el método raíz—, independientemente de dónde esté situado el método en el programa (en este caso lo situé deliberadamente al final). Las sentencias se ejecutan de una en una, en orden, hasta que alcanzan la llamada a un método. Las llamadas a métodos son como un desvío en el flujo de ejecución. En lugar de ir a la siguiente sentencia, te diriges a la primera línea del método invocado, ejecutas todas las sentencias que hay allí, y luego vuelves de nuevo al lugar en el que lo dejaste.

Esto suena bastante sencillo, excepto que tiene recordar que un método puede llamar a otro. Por tanto, mientras estamos en la mitad de `make`, tenemos que dejarlo y ejecutar las sentencias de `tres_lineas`. Pero cuando estamos ejecutando `tres_lineas`, somos interrumpidos tres veces para ir a ejecutar `nueva_linea`.

Por su parte, `nueva_linea` invoca el método predefinido `print`, lo que causa otro desvío. Afortunadamente, Eiffel es lo bastante diestro como para llevar la cuenta de por dónde va, por lo que cuando `print` termina, continúa donde lo dejó en `nueva_linea`, y luego vuelve a `tres_lineas`, y finalmente regresa a `make` de forma que el programa puede terminar.

¿Cuál es la moraleja de este sórdido cuento? Cuando leas un programa, no lo hagas de arriba abajo. En su lugar, sigue el flujo de ejecución.

### 3.8. Parámetros y argumentos

Muchos de los métodos que hemos usado hasta ahora actúan sobre un objeto, siguiendo la sintaxis de poner el nombre del método a continuación del valor, y separado por un punto. Así, para calcular el coseno de un ángulo, se escribe el ángulo y a continuación se añade el sufijo `.cos`. De esta forma, se sabe a qué ángulo hay que calcularle el coseno.

Uno de los métodos predefinidos que hemos usado tiene **parámetros**, que son valores que tienes que proporcionar al método para hacer su trabajo. Por ejemplo, para imprimir una cadena, tienes que proporcionar la cadena entre paréntesis a continuación del nombre del método, que en este caso es `print`.

Existen métodos que necesitan una mezcla de ambas sintaxis. Por ejemplo, para elevar un número a una potencia, se usa el método `pow`, de esta manera:

```
class RAIZ

creation make

feature
  make is
    local
```

```

        x, y: DOUBLE
    do
        x := 2
        y := x.pow(3)
        print(y)
    end
end

```

Este programa calcula  $2^3$ , o sea, 8, y lo muestra por pantalla. Observa que en este caso se usa `pow` de la siguiente forma:

1. Se le pide a `x` que calcule cuánto vale él mismo elevado a un cierto exponente.
2. `pow` recibe el exponente en cuestión como un parámetro.

Si queremos definir un método que acepte parámetros, deberemos especificar no sólo cuántos parámetros tiene, sino además el tipo de cada parámetro. Así que no debería ser sorprendente el hecho de que cuando escribes una definición de método, la lista de parámetros indica el tipo de cada parámetro. Por ejemplo:

```

imprimir_dos_veces(pepe: STRING) is
do
    print(pepe)
    print("%N")
    print(pepe)
end

```

Este método tiene un único parámetro, llamado `pepe`, que es de tipo cadena (`STRING`). Cualquiera que sea el valor del parámetro (y en este momento no tenemos ni idea de cuál es), éste se imprimirá dos veces. Le he llamado `pepe` para sugerirte que el nombre que le das al parámetro es cosa tuya, aunque en general preferirás escoger algo más ilustrativo que `pepe`.

Para poder invocar este método, debemos proporcionarle una cadena. Por ejemplo, podríamos tener un método `make` como este:

```

make is
do
    imprimir_dos_veces("¡No me hagas decírtelo dos veces!")
end

```

La cadena que has proporcionado se llama **argumento**, y decimos que **pasamos** el argumento al método. En este caso hemos creado un valor cadena que contiene el texto “¡No me hagas decírtelo dos veces!” y lo hemos pasado como argumento a `imprimir_dos_veces`, donde, contrariamente a nuestros deseos, se imprimirá dos veces.

Como alternativa, si tuviéramos una variable de tipo `STRING`, podríamos usarla como argumento:

```

make is
local
    argumento: STRING
do
    argumento := "Nunca digas nunca."
    imprimir_dos_veces(argumento)
end

```

Observa aquí algo muy importante: el nombre de la variable que pasamos como argumento (**argumento**) no tiene nada que ver con el nombre del parámetro (**pepe**). Déjame que lo diga de nuevo:

**El nombre de la variable que pasamos como argumento no tiene nada que ver con el nombre del parámetro.**

Puede ser el mismo o puede ser distinto, pero es importante darse cuenta de que no son la misma cosa, excepto el hecho de que tienen el mismo valor (en este caso la cadena "Nunca digas nunca.>").

El valor que proporcionas como argumento debe tener el mismo tipo que el parámetro del método al que llamas. Esta regla es muy importante, pero a menudo se torna complicada en Eiffel por dos razones:

- Hay algunos métodos que aceptan argumentos de tipos muy diferentes. Por ejemplo, puedes pasar un argumento de *cualquier* tipo a **print**, y éste hará correctamente su trabajo. Esto es una excepción, por otra parte (otra más de las que tiene el método **print**).
- Si violas esta regla, el compilador suele generar un confuso mensaje de error. En lugar de decir algo como "Estás pasando un tipo de argumento incorrecto a este método", probablemente diga algo sobre que no puede encontrar un método con ese nombre que pueda aceptar un argumento de ese tipo. Una vez que has visto este mensaje de error unas cuantas veces, sin embargo, te harás una idea de cómo interpretarlo.

Una última cosa que debes comprender es que los parámetros y otras variables sólo existen dentro de sus propios métodos. En el interior de **make** no existe **pepe**. Si intentas usarlo, el compilador se quejará. Igualmente, dentro de **imprimir\_dos\_veces** no existe **argumento**.

### 3.9. Métodos con varios parámetros

En ocasiones podemos necesitar definir métodos que usen varios parámetros. La sintaxis para ello es sencilla. Por ejemplo:

```
imprimir_hora(hora, minutos: INTEGER) is
do
    print(hora)
    print(":")
    print(minutos)
end
```

Una típica fuente de confusión es que no necesitas declarar el tipo de los argumentos. ¡Lo siguiente es incorrecto!

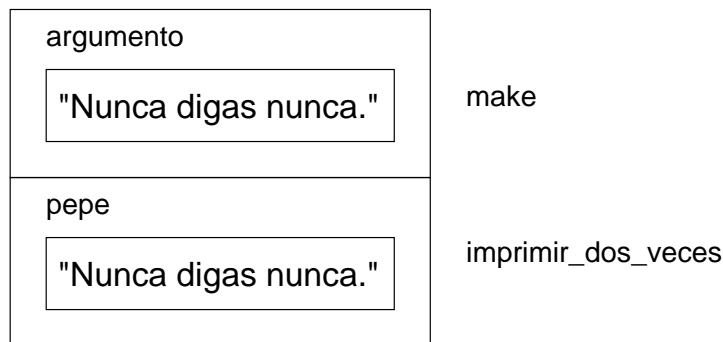
```
local
    hora, minutos: INTEGER
do
    hora := 11
    minutos := 59
    imprimir_hora(hora, minutos: INTEGER)  -- <INCORRECTO!
end
```

En este caso, Eiffel puede determinar el tipo de **hora** y **minutos** mirando sus declaraciones. Es innecesario e incorrecto incluir el tipo cuando los pasas como argumentos. La sintaxis correcta es **imprimir\_hora(hora, minutos)**.

### 3.10. Diagramas de pila

Para llevar la cuenta sobre qué variables pueden ser usadas y dónde, a veces es útil dibujar un diagrama de pila. Como los diagramas de estado, los diagramas de pila muestran el valor de cada variable, pero además muestra el método al que pertenece cada variable.

Cada método se representa por un marco. Un marco es una caja con el nombre del método a un lado y los parámetros y variables del método dentro. El diagrama de pila para el ejemplo anterior se parece a esto:



El orden de la pila muestra el flujo de ejecución. En este caso, `imprimir_dos_veces` fue llamado por `make`.

Dentro del marco de `make` encontramos la variable local `argumento`, cuyo valor en el momento de dibujar el diagrama era “Nunca digas nunca.”. De igual forma, dentro del marco de `imprimir_dos_veces` vemos el parámetro `pepe`, cuyo valor (lógicamente) coincide con el anterior, “Nunca digas nunca.”.

Si se encuentra en error durante la llamada a un método, Eiffel muestra el nombre del método, y el nombre del método que lo llamó, y el nombre del método que a su vez llamó a éste, y así sucesivamente hasta llegar a `make`.

Como podrás observar, lo que se hace es recorrer la pila de abajo arriba, mostrando cada vez el nombre del método al que corresponde cada marco.

### 3.11. Métodos con resultados

Habrás podido darte cuenta que algunos de los métodos que hemos usado, como los métodos matemáticos, proporcionan resultados. Otros métodos, como `print` o `nueva_linea`, realizan alguna acción pero no devuelven ningún valor. Esto plantea varias cuestiones:

- ¿Qué ocurre si invocas a un método y no haces nada con el resultado (p.e. no lo asignas a una variable o lo usas como parte de una expresión mayor)?
- ¿Qué ocurre si usas un método `print` como parte de una expresión, como `print(";boo!") + 7`?
- ¿Podemos escribir métodos que proporcionen resultados, o tenemos que atrancarnos con cosas como `nueva_linea` e `imprimir_dos_veces`?

La respuesta a la tercera pregunta es “sí, puedes escribir métodos que devuelvan valores”, y lo haremos en un par de capítulos. Dejaré que tú mismo respondas a las otras dos preguntas haciendo pruebas. De hecho, cada vez que tengas una pregunta sobre qué es legal o ilegal en Eiffel, una buena forma de encontrar la respuesta es preguntarle al compilador.

### 3.12. Glosario

**real:** Un tipo de variable (o valor) que puede contener números con parte decimal además de enteros. En Eiffel a este tipo se le llama `DOUBLE`.

**clase:** Una colección de métodos etiquetada con un nombre. Hasta ahora, hemos escrito las clases llamadas `HOLA` y `NUEVA_LINEA`.

**método:** Una secuencia de sentencias, etiquetada con un nombre, que realiza alguna función útil. Los métodos pueden o no tener parámetros, y pueden o no producir un resultado.

**parámetro:** Un trozo de información que proporcionas a la hora de declarar a un método. Los parámetros son como variables en el sentido de que contienen valores y tienen tipos.

**argumento:** Un valor que proporcionas cuando llamas a un método. Este valor debe tener el mismo tipo que su correspondiente parámetro.

**invocar:** Hacer que un método se ejecute.

## Capítulo 4

# Alternativas y recursividad

### 4.1. Los operadores módulo y división entera

El operador “módulo” opera sobre enteros (y expresiones enteras) y devuelve el *resto* de dividir el primer operando entre el segundo. En Eiffel, el operador módulo se representa por `\|`. El operador “división entera” devuelve la parte entera del resultado de dividir el primer operando entre el segundo. En Eiffel se representa por `//`. La sintaxis es exactamente la misma que para los demás operadores:

```
local
  cociente, resto: INTEGER
do
  cociente := 7 // 3
  resto := 7 \| 3
end
```

El primer operador, división entera, devuelve 2. El segundo operador devuelve 1. Así, 7 dividido entre 3 da 2, y de resto 1.

El operador módulo resulta ser sorprendentemente útil. Por ejemplo, puedes comprobar si un número es divisible entre otro: si `x \| y` da cero, entonces `x` es divisible entre `y`.

Además, puedes usar el operador módulo para extraer el dígito o dígitos más a la derecha de un número. Por ejemplo `x \| 10` devuelve el dígito más a la derecha de `x` (en base 10). De igual forma `x \| 100` devuelve los últimos dos dígitos.

### 4.2. Ejecución alternativa

Con vías a escribir programas útiles, casi siempre necesitaremos la capacidad de comprobar ciertas condiciones y de cambiar el comportamiento del programa de acuerdo a dicha comprobación. Las **sentencias alternativas** nos proporcionan esa capacidad. La forma más simple es la sentencia `if`:

```
if x > 0 then
  print("x es positivo")
end
```



La expresión que se encuentra entre las palabras `if` y `then` se denomina condición. Si es verdadera, entonces se ejecutarán las sentencias situadas entre las palabras `then` y `end`. Si la condición es falsa, no ocurre nada.

La condición puede contener cualquier operador de comparación, a veces llamados **operadores relacionales**:

<code>x = y</code>	-- 'x' es igual a 'y'
<code>x /= y</code>	-- 'x' no es igual a 'y'
<code>x &gt; y</code>	-- 'x' es mayor que 'y'
<code>x &lt; y</code>	-- 'x' es menor que 'y'
<code>x &gt;= y</code>	-- 'x' es mayor o igual que 'y'
<code>x &lt;= y</code>	-- 'x' es menor o igual que 'y'

Aunque estas operaciones probablemente te sean familiares, la sintaxis que usa Eiffel es un poco diferente del aspecto de los símbolos matemáticos como  $=$ ,  $\neq$  y  $\leq$ . En cualquier caso, los operadores de Eiffel se escogieron con la intención de parecerse lo más posible a los operadores matemáticos usuales. Recuerda, además, que no existen  $=<$  ni  $=>$ .

Las dos partes de un operador relacional deben ser del mismo tipo. Sólo puedes comparar `INTEGERs` con `INTEGERs` y `DOUBLEs` con `DOUBLEs`. ¡Desgraciadamente, por ahora no puedes comparar `STRINGs`! Hay una forma de comparar cadenas, pero no la tendremos en cuenta hasta dentro de un par de capítulos.

### 4.3. Ejecución alternativa doble

Una segunda forma de ejecución condicional es la ejecución alternativa doble, en la cual existen dos posibilidades, y la condición determina cuál se ejecuta. La sintaxis tiene este aspecto:

```
if x \ 2 = 0 then
    print("x es par")
else
    print("x es impar")
end
```

Si el resto de dividir `x` entre 2 es cero, entonces sabemos que `x` es par, y este código imprime un mensaje a tal efecto. Si la condición es falsa, se ejecuta el segundo conjunto de sentencias. Como la condición debe ser verdadera o falsa, se ejecutará exactamente una de las alternativas.

Al margen de esto, si crees que puedes necesitar comprobar la paridad (par o impar) de números a menudo, podrías querer “envolver” este código en un método, como sigue:

```
imprimir_paridad(x: INTEGER) is
do
    if x \ 2 = 0 then
        print("x es par")
    else
        print("x es impar")
    end
end
```

Ahora tienes un método llamado `imprimir_paridad` que mostrará un mensaje apropiado por cada entero que le proporciones. En `make` podrías llamar a ese método así:

```
imprimir_paridad(17)
```

Recuerda siempre que cuando *llamas* a un método, no tienes que declarar los tipos de los argumentos que proporcionas. Eiffel se da cuenta de qué tipos son. Debes resistir la tentación de escribir cosas como ésta:

```
local
  numero: INTEGER
do
  numero := 17
  imprimir_paridad(numero: INTEGER) -- ¡¡¡INCORRECTO!!!
end
```

## 4.4. Alternativas encadenadas

A veces queremos comprobar un número de condiciones relacionadas y elegir una entre varias acciones. Una manera de hacer esto es **encadenando** una serie de `ifs` y `elses`, uniendo estas dos palabras en una nueva palabra clave llamada `elseif`:

```
if x > 0 then
  print("x es positivo")
elseif x < 0 then      -- observa que 'elseif' va junto
  print("x es negativo")
else
  print("x es cero")
end
```

Estos encadenamientos pueden ser tan largos como quieras, aunque pueden ser difíciles de leer si se te van de las manos. Una manera de hacer que sean más fáciles de leer es usar una indentación estándar, como se ha mostrado en estos ejemplos. Si mantienes todas las sentencias y palabras clave alineadas, serás menos propenso a cometer errores y podrás encontrarlos más rápidamente si aparecen.

## 4.5. Alternativas anidadas

Además del encadenamiento, también puedes anidar una alternativa dentro de otra. Podríamos haber escrito el ejemplo anterior como:

```
if x = 0 then
  print("x es cero")
else
  if x > 0 then
    print("x es positivo")
  else
    print("x es negativo")
  end -- del 'if x > 0 ...'
end -- del 'if x = 0 ...'
```

Ahora hay una alternativa exterior que contiene dos ramas. La primera rama contiene una sola sentencia `print`, pero la segunda rama contiene otra sentencia alternativa, la cual tiene a su vez

otras dos ramas. Afortunadamente, las dos ramas son sentencias `print`, aunque podrían haber sido sentencias alternativas también. Cada sentencia alternativa acaba con la palabra clave `end`, y es por eso por lo que tenemos dos `ends`, una para cada `if`.

Observa de nuevo que la indentación ayuda a mantener clara la estructura, pero no obstante, las alternativas anidadas se tornan difíciles de leer muy rápidamente. En general, es una buena idea evitarlas siempre que se pueda.

Por otra parte, este tipo de **estructura anidada** es común, y la veremos de nuevo, por lo que es mejor que sepas usarla.

## 4.6. Recursividad

Mencioné en el último capítulo que está permitido el que un método llame a otro, y hemos visto varios ejemplos de ellos. No mencioné que también está permitido el que un método se llame a sí mismo. Puede que no sea obvio el por qué eso es bueno, pero resulta ser una de las cosas más mágicas e interesantes que un programa puede realizar.

Por ejemplo, mira el siguiente método:

```

cuenta_atras(n: INTEGER) is
do
  if n = 0 then
    print("¡Lanzamiento!")
  else
    print(n) print("%N")
    cuenta_atras(n-1)
  end
end

```

El nombre del método es `cuenta_atras` y recibe un único entero como parámetro. Si el parámetro es cero, imprime la palabra “¡Lanzamiento!”. En otro caso, imprime el número y luego llama al método llamado `cuenta_atras` —él mismo— pasando `n-1` como argumento.

¿Qué ocurre si llamamos al método, desde `make`, de esta forma?

```
cuenta_atras(3)
```

Comienza la ejecución de `cuenta_atras` con `n=3`, y como `n` no es cero, imprime el valor 3, y luego se llama a sí mismo...

Comienza la ejecución de `cuenta_atras` con `n=2`, y como `n` no es cero, imprime el valor 2, y luego se llama a sí mismo...

Comienza la ejecución de `cuenta_atras` con `n=1`, y como `n` no es cero, imprime el valor 1, y luego se llama a sí mismo...

Comienza la ejecución de `cuenta_atras` con `n=0`, y como `n` es cero, imprime la palabra “¡Lanzamiento!”, y luego finaliza.

Finaliza la `cuenta_atras` que tenía `n=1`.

Finaliza la `cuenta_atras` que tenía `n=2`.

Finaliza la `cuenta_atras` que tenía `n=3`.

Y luego vuelves a `make` (vaya viaje). Así que la salida final se parecerá a esto:

```

3
2
1
;Lanzamiento!

```

Como segundo ejemplo, veamos de nuevo los métodos `nueva_linea` y `tres_lineas`.

```

nueva_linea is
do
    print("%N")
end

tres_lineas is
do
    nueva_linea
    nueva_linea
    nueva_linea
end

```

Aunque esto funciona, no sería de mucha ayuda si quisieras imprimir 2 nuevas líneas, o 106. Una alternativa mejor podría ser

```

n_lineas(n: INTEGER) is
do
    if n > 0 then
        print("%N")
        n_lineas(n-1)
    end
end

```

Este programa es muy similar; mientras `n` sea mayor que cero, imprimirá una nueva línea y luego se llamará a sí mismo para imprimir `n-1` nuevas líneas adicionales. Por tanto, el número total de nuevas líneas que se imprimirán será  $1 + (n-1)$ , que resulta ser `n`.

El proceso por el cual un método se llama a sí mismo se denomina **recursividad**, y dicho método se dice que es **recursivo**.

## 4.7. Diagramas de pila para métodos recursivos

En el capítulo anterior usamos un diagrama de pila para representar el estado de un programa durante una llamada a un método. El mismo tipo de diagrama puede hacer que un método recursivo sea más fácil de interpretar.

Recuerda que cada vez que se invoca a un método se crea una nueva instancia del método que contiene una nueva versión de las variables locales y los parámetros del método.

La figura siguiente es un diagrama de pila para `cuenta_atras`, invocado con `n=3`:

	<code>make</code>
<code>n: 3</code>	<code>cuenta_atras</code>
<code>n: 2</code>	<code>cuenta_atras</code>
<code>n: 1</code>	<code>cuenta_atras</code>
<code>n: 0</code>	<code>cuenta_atras</code>

Hay una instancia de `make` y cuatro instancias de `cuenta_atras`, cada una con un valor diferente en el parámetro `n`. El fondo de la pila, `cuenta_atras` con `n=0`, es el caso base. No hace ninguna llamada recursiva, por lo que no hay más instancias de `cuenta_atras`.

La instancia de `make` está vacía porque `make` no tiene parámetros ni variables locales. Como ejercicio, dibuja un diagrama de pila para `n_lineas`, invocada con el parámetro `n=4`.

## 4.8. Glosario

**módulo:** Un operador que trabaja sobre enteros y devuelve el resto de dividir uno entre otro. En Eiffel se representa por el símbolo `//`.

**alternativa:** Un bloque de sentencias que pueden o no ser ejecutadas dependiendo de alguna condición.

**encadenamiento:** Una forma de unir varias sentencias condicionales en secuencia.

**anidamiento:** Colocar una sentencia condicional dentro de una o más ramas de otra sentencia condicional.

**interface:** Una descripción de los parámetros que requiere un método, y los tipos de esos parámetros.

**recursividad:** El proceso de invocar el mismo método que se está actualmente ejecutando.

**recursión infinita:** Un método que se llama a sí mismo recursivamente sin alcanzar nunca el caso base. Lo normal es que acabe desbordando la pila y provocando la parada del programa.

## Capítulo 5

# Métodos función

### 5.1. Valores de retorno

Algunos de los métodos predefinidos que hemos usado, como los métodos matemáticos, producen resultados. Esto es, el efecto de llamar al método es generar un nuevo valor, el cual normalmente se asigna a una variable o se usa como parte de una expresión. Por ejemplo:

```
local
  e, ancho: DOUBLE
do
  e := (1.0).exp
  ancho := radio * (angulo).cos
end
```

Pero hasta ahora todos los métodos que hemos escrito han sido métodos **procedimiento**; esto es, métodos que no devuelven ningún valor. Cuando llamas a un método procedimiento, normalmente se ocupa una línea para él mismo, sin ninguna asignación:

```
n_lineas(3)
imprimir_dos_veces("Hola")
```

En este capítulo, vamos a escribir métodos que devuelven cosas, a los que llamaré métodos **función**. El primer ejemplo es **area**, que recibe un **DOUBLE** como parámetro, y devuelve el área de un círculo dado su radio:

```
area(radio: DOUBLE): DOUBLE is
do
  Result := Pi * radio * radio
end
```

La primera cosa que habrás observado es que el comienzo inicio de la definición del método es diferente. Ahora, además del nombre del método y la lista de parámetros entre paréntesis, se incluye también el tipo del valor de retorno del método, que en este caso es **DOUBLE**. Fíjate que la sintaxis para declarar el tipo de retorno recuerda mucho a la manera de declarar el tipo de variables y parámetros: el nombre de lo que se declara, dos puntos, y el tipo de lo que se declara.

Observa, además, que la única línea del método contiene una asignación en la que se almacena el valor de retorno (el resultado de **Pi \* radio \* radio**) en una variable especial llamada **Result**. Esa variable es especial por tres motivos:

1. No hay que declararla: se puede utilizar directamente.
2. Su tipo coincide siempre con el tipo de retorno del método.
3. El valor que almacenemos en ella será el valor de retorno del método.

Por tanto, cuando el método finalice su ejecución, porque haya alcanzado su última instrucción, el valor de retorno que devolverá al método que lo llamó será aquél que se encuentre almacenado en la variable especial **Result**.

Aquí tenemos otro ejemplo. En este caso, el valor de retorno dependerá de una condición:

```
valor_absoluto(x: DOUBLE): DOUBLE is
do
  if x < 0 then
    Result := -x
  else
    Result := x
  end
end
```

Si el valor del parámetro es menor que cero, el resultado será el valor del parámetro cambiado de signo. En caso contrario, se devolverá el mismo valor que el recibido como parámetro.

Es importante hacer notar que la ejecución del método no finaliza cuando se asigna el valor a la variable **Result**. Una asignación a la variable **Result** es como cualquier otra asignación. La ejecución de un método sólo puede acabar cuando el flujo de ejecución alcanza la última línea de dicho método.

Si el resultado de un método función depende de una condición, entonces debes garantizar que *cada posible camino* a través del programa dará un valor adecuado a **Result**. Por ejemplo:

```
valor_absoluto(x: DOUBLE): DOUBLE is
  -- ¡¡ INCORRECTO !!
do
  if x < 0 then
    Result := -x
  else
    if x > 0 then
      Result := x
    end
  end
end
```

Este programa no es correcto porque si *x* resulta ser 0, entonces ninguna de las condiciones serán verdaderas y el método terminará sin asignar un valor correcto a **Result**.

¿Qué valor se devolverá entonces cuando se retorne al punto en el que se llamó al método? Al no haberse asignado ningún valor a **Result**, se devolverá el valor por defecto correspondiente al tipo del método. Como en este caso el método devuelve un valor de tipo **DOUBLE**, se devolvería el valor por defecto de **DOUBLE**, que es cero. Podrías pensar que *ese* era precisamente el comportamiento que buscábamos: si *x* es cero, su valor absoluto también es cero. No obstante, ese no es el caso general, y si no quieres encontrarte luego con errores extraños y difíciles de encontrar, exígete a ti mismo el que la variable **Result** tenga siempre un valor asignado antes de acabar la ejecución del método correspondiente.

## 5.2. Desarrollo de programas

En este punto ya deberías ser capaz de leer métodos completos escritos en Eiffel y decir lo que hacen. Pero puede no serte claro todavía cómo llegar a escribirlos. Voy a sugerirte una técnica que he llamado **desarrollo incremental**.

Como ejemplo, imagina que quieres encontrar la distancia entre dos puntos, dados por las coordenadas  $(x_1, y_1)$  y  $(x_2, y_2)$ . Por su definición usual,

$$\text{distancia} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (5.2.1)$$

El primer paso es considerar qué aspecto debería tener en Eiffel un método `distancia`. En otras palabras, cuáles son sus entradas (parámetros) y cuál es la salida (valor de retorno).

En este caso, los dos puntos son los parámetros, y es natural representarlos usando cuatro `DOUBLES`. El valor de retorno es la distancia, que tendrá un tipo `DOUBLE`.

Ya podemos escribir un boceto del método:

```
distancia(x1, y1, x2, y2: DOUBLE): DOUBLE is
do
    Result := 0.0
end
```

La sentencia `Result := 0.0` es necesaria para compilar el programa. Evidentemente, en esta etapa el programa no hace nada útil, pero vale la pena intentar compilarlo de manera que podamos identificar cualquier error sintáctico antes de que hagamos algo más complicado.

Para probar el nuevo método, tenemos que invocarlo con valores de ejemplo. En alguna parte de `make` añadimos:

```
dist := distancia(1.0, 2.0, 4.0, 6.0)
```

Elegí esos valores porque la distancia horizontal es 3 y la distancia vertical es 4; de esta forma, el resultado será 5 (la hipotenusa de un triángulo de lados 3, 4, 5). Cuando pruebas un método, es útil conocer de antemano la respuesta correcta.

Una vez que hemos comprobado la sintaxis de la definición del método, podemos comenzar a añadir líneas de código, una cada vez. Tras cada cambio incremental, recompilamos y ejecutamos el programa. De esta forma, en cualquier momento sabemos exactamente dónde debe estar el error —en la última línea que hemos añadido.

El siguiente paso en el cálculo es encontrar las diferencias  $x_2 - x_1$  y  $y_2 - y_1$ . Almacenaré esos valores en variables temporales llamadas `dx` y `dy`.

```
distancia(x1, y1, x2, y2: DOUBLE): DOUBLE is
local
    dx, dy: DOUBLE
do
    dx := x2 - x1
    dy := y2 - y1
    print("dx es ")
    print(dx)
    print("%N")
    print("dy es ")
    print(dy)
    Result := 0.0
end
```



Añadí sentencias de impresión que me permitirán comprobar los valores intermedios antes de proceder. Como ya mencioné, yo ya sé que deberían ser 3.0 y 4.0.

Cuando el método esté finalizado quitaré las sentencias de impresión. Este tipo de código se llama **andamiaje**, porque ayuda a construir el programa, pero no es parte del producto final. A veces es una buena idea mantener el andamiaje, pero dentro de un comentario, por si se da el caso de que lo necesites más tarde.

El siguiente paso en el desarrollo es el cuadrado de  $dx$  y  $dy$ . Podríamos usar el método `pow`, pero es más simple y rápido multiplicar simplemente cada término por sí mismo.

```
distancia(x1, y1, x2, y2: DOUBLE): DOUBLE is
  local
    dx, dy, ds_al_cuadrado: DOUBLE
  do
    dx := x2 - x1
    dy := y2 - y1
    ds_al_cuadrado = dx*dx + dy*dy
    print("ds_al_cuadrado es ")
    print(ds_al_cuadrado)
    Result := 0.0
  end
```

De nuevo, debería compilar y ejecutar el programa en esta etapa y comprobar el valor intermedio (que debería ser 25.0).

Finalmente, usaremos el método `sqrt` para calcular y devolver el resultado.

```
distancia(x1, y1, x2, y2: DOUBLE): DOUBLE is
  local
    dx, dy, ds_al_cuadrado: DOUBLE
  do
    dx := x2 - x1
    dy := y2 - y1
    ds_al_cuadrado := dx*dx + dy*dy
    Result := ds_al_cuadrado.sqrt
  end
```

Después, en `make`, deberíamos imprimir y comprobar el valor del resultado.

A medida que vayas ganando más experiencia como programador, te irás viendo a ti mismo escribiendo y depurando más de una línea a la vez. No obstante, este proceso de desarrollo incremental puede ahorrarte un montón de tiempo de depuración.

Los aspectos clave del proceso son:

- Comenzar con un programa que funcione y hacer pocos e incrementales cambios. En cualquier momento, si hay un error, sabrás exactamente dónde está.
- Usa variables temporales para almacenar valores intermedios de manera que puedas imprimirlos y comprobarlos.
- Una vez que el programa esté funcionando, podrás querer eliminar algo del andamiaje o unir varias sentencias en expresiones compuestas, pero sólo si eso no hace al programa difícil de leer.

### 5.3. Composición

Como podrías esperar, una vez que defines un nuevo método, puedes usarlo como parte de una expresión, y puedes construir nuevos métodos usando métodos existentes. Por ejemplo, ¿qué pasa si alguien te da dos puntos, el centro de un círculo y un punto de su perímetro, y te pide que calcules el área del círculo?

Digamos que el punto central está almacenado en las variables `xc` y `yc`, y el punto del perímetro está en `xp` y `yp`. El primer paso es encontrar el radio del círculo, que es la distancia entre los dos puntos. Afortunadamente, tenemos un método, `distancia`, que hace eso.

```
r := distancia(xc, yc, xp, yp)
```

El segundo paso es encontrar el área del círculo con ese radio, y devolverlo.

```
a := area(r)
Result := a
```

Envolviendo todo en un método, tenemos:

```
area2(xc, yc, xp, yp: DOUBLE): DOUBLE is
  local
    r, a: DOUBLE
  do
    r := distancia(xc, yc, xp, yp)
    a := area(r)
    Result := a
  end
```

Como ya tenemos un método llamado `area`, no podemos usar ese nombre para nuestro nuevo método, por lo que lo llamamos (por ejemplo) `area2`. Existen lenguajes de programación que admiten el que dos métodos diferentes tengan el mismo nombre, distinguiéndose por tanto en el número y tipo de sus parámetros. Eiffel no es uno de tales lenguajes.

Las variables temporales `r` y `a` son útiles durante el desarrollo y la depuración, pero una vez que el programa funciona podemos hacer que quede más conciso componiendo las llamadas a los métodos:

```
area2(xc, yc, xp, yp: DOUBLE): DOUBLE is
  do
    Result := area(distancia(xc, yc, xp, yp))
  end
```

### 5.4. Expresiones lógicas

Muchas de las operaciones que hemos visto producen resultados que son del mismo tipo que sus operandos. Por ejemplo, el operador `+` toma dos `INTEGERs` y produce un `INTEGER`, o dos `DOUBLEs` y produce un `DOUBLE`, etc.

Las excepciones que hemos visto son los **operadores relacionales**, que comparan enteros y reales y devuelven verdadero (`True`) o falso (`False`). `True` y `False` son valores especiales en Eiffel, y juntos forman un tipo llamado **lógico** o booleano (`BOOLEAN`). Recordarás que cuando definimos lo que era un tipo, dijimos que era un conjunto de valores. En el caso de `INTEGERs`, `DOUBLEs` y `STRINGs`, esos conjuntos son bastante grandes. Para los `BOOLEANs`, no lo son tanto.

Las expresiones y variables lógicas funcionan justamente igual que otros tipos de variables y expresiones:

```
local
  pepe, resultado: BOOLEAN
do
  pepe := True
  resultado := False
end
```

Las variables `pepe` y `resultado` son variables lógicas, puesto que así queda reflejado en su declaración (lo que hay entre `local` y `do`). A cada variable se le asigna un valor lógico usando las palabras clave `True` y `False`, que representan a los dos únicos posibles valores del tipo lógico.

Como ya mencioné, el resultado de un operador relacional es un valor lógico, por lo que se puede almacenar el resultado de la comparación en una variable:

```
local
  par, positivo: BOOLEAN
do
  par := (n \ 2 = 0)      -- verdadero si n es par
  positivo := (x > 0)     -- verdadero si x es positivo
end
```

y posteriormente usarlo como parte de una sentencia alternativa:

```
if par then
  print("n era par cuando lo comprobé")
end
```

A una variable usada de esta forma se la denomina frecuentemente **indicador**, pues los indican la presencia o ausencia de alguna condición.

## 5.5. Operadores lógicos

Existen tres **operadores lógicos** en Eiffel: `and` (Y), `or` (O) y `not` (NO). La semántica (significado) de estos operadores es similar a su significado en castellano. Por ejemplo `x > 0 and x < 10` es verdadero sólo si `x` es mayor que cero Y menor que 10.

`par or n \ 3 = 0` es verdadero si *alguna* de las condiciones es verdadera, es decir, si `par` es verdadero O el número es divisible entre 3.

Finalmente, el operador `not` tiene el efecto de negar o invertir una expresión lógica, por lo que `not par` es verdadero si `par` es falso —si el número es impar.

Los operadores lógicos a menudo proporcionan una manera de simplificar sentencias alternativas anidadas. Por ejemplo, ¿cómo escribirías el siguiente código usando una sola sentencia alternativa?

```
if x > 0 then
  if x < 10 then
    print("x es un único dígito positivo.")
  end
end
```

## 5.6. Métodos lógicos

Los métodos pueden devolver valores lógicos igual que cualquier otro tipo, lo que a menudo es conveniente para ocultar comprobaciones complejas dentro de los métodos. Por ejemplo:

```
es_un_solo_digito(x: INTEGER): BOOLEAN is
do
    if x >= 0 and x < 10 then
        Result := True
    else
        Result := False
    end
end
```

El nombre de este método es `es_un_solo_digito`. Es común darle a los métodos lógicos un nombre que suene a pregunta sí/no. El tipo de retorno es `BOOLEAN`, lo que significa que a la variable especial `Result` hay que asignarle una expresión lógica.

El código en sí mismo está claro, aunque es un poco más grande de lo que se necesitaría. Recuerda que la expresión `x >= 0 and x < 10` tiene tipo lógico, por lo que no hay nada incorrecto en devolverlo directamente, y evitar de paso la sentencia `if`:

```
es_un_solo_digito(x: INTEGER): BOOLEAN is
do
    Result := x >= 0 and x < 10
end
```

En `make` puedes invocar a este método de las formas ya usuales:

```
local
    indicador: BOOLEAN
do
    indicador := not es_un_solo_digito(17)
    print(es_un_solo_digito(2))
end
```

La primera sentencia asigna el valor `True` a `indicador` sólo si `17` *no* es un número de un sólo dígito. La segunda sentencia imprime `True` porque `2` es un número de un sólo dígito. Y sí, `print` también sabe imprimir valores lógicos.

El uso más común de los métodos lógicos es dentro de sentencias alternativas

```
if es_un_solo_digito(x) then
    print("x es pequeño")
else
    print("x es grande")
end
```

## 5.7. Más recursividad

Ahora que tenemos métodos que devuelven valores, podrías estar interesado en saber que tenemos un lenguaje de programación **completo**, en el sentido de que todo lo que pueda calcularse

puede ser expresado en este lenguaje. Todo programa que se haya escrito alguna vez, puede ser reescrito usando las características de lenguaje que hemos usado hasta ahora (en realidad, podríamos necesitar unas pocas acciones para controlar dispositivos como el teclado, ratón, discos, etc., pero eso es todo).

Demostrar que esta afirmación es cierta es un ejercicio nada trivial realizado por primera vez por Alan Turing, uno de los primeros informáticos (bueno, algunos afirman que era matemático, pero muchos de los primeros informáticos comenzaron como matemáticos). Por este motivo, se le conoce como la tesis de Turing. Si acudes a un curso de Teoría de la Computación, tendrás ocasión de ver la demostración.

Para darte una idea de lo que puedes hacer con las herramientas que hemos aprendido hasta ahora, veamos algunos métodos para evaluar funciones matemáticas definidas recursivamente. Una definición recursiva es similar a una definición circular, en el sentido de que la definición contiene una referencia a lo que se está definiendo. Una definición verdaderamente circular no es generalmente muy útil:

**frabuloso:** un adjetivo usado para describir algo que es frabuloso.

Si ves esa definición en el diccionario, te podrías sentir fastidiado. Por otra parte, si buscas la definición de la función matemática **factorial**, puede que te encuentres con algo como ésto:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

(El factorial se representa normalmente con el símbolo !). Esta definición dice que el factorial de 0 es 1, y que el factorial de cualquier otro valor,  $n$ , es  $n$  multiplicado por el factorial de  $n-1$ . Así  $3!$  es 3 por  $2!$ , que es 2 por  $1!$ , que es 1 por  $0!$ . Juntando todo, tenemos que  $3!$  es igual a 3 por 2 por 1 por 1, lo que da 6.

Si puedes escribir una definición recursiva de algo, normalmente puedes escribir un programa Eiffel para evaluarlo. El primer paso es decidir qué parámetros tendrá la función, y cuál será el tipo de retorno. Pensando un poco, concluyes que el factorial recibe un entero como parámetro y devuelve un entero:

```
factorial(n: INTEGER): INTEGER is
do
end
```

Si el argumento resulta ser cero, todo lo que tenemos que hacer es devolver 1:

```
factorial(n: INTEGER): INTEGER is
do
  if n = 0 then
    Result := 1
  end
end
```

En otro caso, y esta es la parte interesante, tenemos que hacer una llamada recursiva para encontrar el factorial de  $n-1$ , y luego multiplicarlo por  $n$ .

```
factorial(n: INTEGER): INTEGER is
local
  recursivo: INTEGER
do
```

```

    if n = 0 then
        Result := 1
    else
        recursivo := factorial(n-1)
        Result := n * recursivo
    end
end

```

Si observamos el flujo de ejecución de este programa, veremos que es similar a `n_lineas`, del capítulo anterior. Si invocamos a `factorial` con el valor 3:

Como 3 no es cero, tomamos la segunda rama y calculamos el factorial de  $n - 1$ ...

Como 2 no es cero, tomamos la segunda rama y calculamos el factorial de  $n - 1$ ...

Como 1 no es cero, tomamos la segunda rama y calculamos el factorial de  $n - 1$ ...

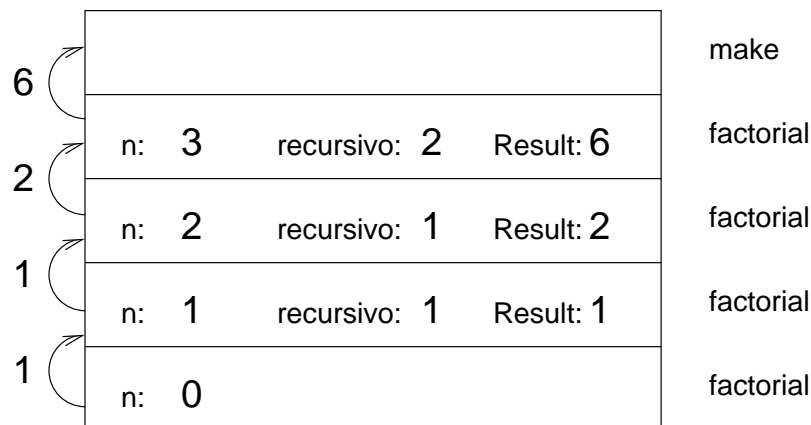
Como 0 *es* cero, tomamos la primera rama y devolvemos el valor 1 inmediatamente, sin hacer nuevas llamadas recursivas.

El valor devuelto (1) se multiplica por `n`, que es 1, y se devuelve el resultado.

El valor devuelto (1) se multiplica por `n`, que es 2, y se devuelve el resultado.

El valor devuelto (2) se multiplica por `n`, que es 3, y el resultado, 6, se devuelve a `make`, o a quien quiera que haya invocado a `factorial(3)`.

Este es el aspecto del diagrama de pila para esta secuencia de llamadas a métodos:



Los valores de retorno se muestran siendo devueltos hacia arriba por la pila.

Observa que en la última instancia de `factorial`, la variables local `recursivo` no tiene asignado un valor porque cuando `n=0` la rama que le asigna uno no se ejecuta.

## 5.8. Salto de fe

Seguir el flujo de ejecución es una manera de leer programas, como como has visto en la sección anterior, puede llegar rápidamente a ser laberíntico. Una alternativa es lo que yo llamo el “salto de fe”. Cuando llegas a la invocación de un método, en lugar de seguir el flujo de ejecución, *supones* que el método funciona correctamente y devuelve el valor apropiado.

De hecho, ya estas practicando este salto de fe cuando usas métodos predefinidos. Cuando llamas a `cos` o `log`, no examinas las implementaciones de estos métodos. Supones que funcionan, porque la gente que escribió esos métodos predefinidos eran buenos programadores.

Bien, lo mismo es cierto cuando llamas a uno de tus métodos. Por ejemplo, en la sección 5.6 escribimos un método llamado `es_un_solo_digito` que determina si un número está entre 0 y 9. Una vez que nos hemos convencido a nosotros mismos de que este método es correcto —probando y examinando el código— podemos usar el método sin tener que mirar el código de nuevo.

Lo mismo es cierto para los programas recursivos. Cuando tienes una llamada recursiva, en lugar de seguir el flujo de ejecución, deberías *suponer* que la llamada recursiva funciona (proporciona el resultado correcto), y luego preguntarte a ti mismo, “suponiendo que calcular encontrar el factorial de  $n - 1$ , ¿puedo calcular el factorial de  $n$ ?” En este caso, está claro que puedes, multiplicándolo por  $n$ .

Por supuesto, es un poco extraño suponer que el método funciona correctamente cuando todavía no lo hemos terminado de escribir, pero ¡por eso se llama un salto de fe!

## 5.9. Un ejemplo más

En el ejemplo anterior usé variables temporales para reflejar los pasos a dar, y hacer el código más fácil de depurar, pero podría haber ahorrado unas cuantas líneas:

```
factorial(n: INTEGER): INTEGER is
do
  if n = 0 then
    Result := 1
  else
    Result := n * factorial(n-1)
  end
end
```

A partir de ahora tenderé a usar la versión más concisa, pero te recomiendo que uses la versión más explícita mientras estés desarrollando código. Una vez que lo tengas funcionando, podrás acortarlo más, si te sientes inspirado.

Después del `factorial`, el siguiente ejemplo clásico de función matemática definida recursivamente es la función de `fibonacci`, que tiene la siguiente definición:

$$\begin{aligned} \text{fibonacci}(0) &= 1 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2) \end{aligned}$$

Traducido a Eiffel, sería

```
fibonacci(n: INTEGER): INTEGER is
do
  if n = 0 or n = 1 then
    Result := 1
  else
    Result := fibonacci (n-1) + fibonacci (n-2)
  end
end
```

Si intentas seguir aquí el flujo de ejecución, incluso para valores muy pequeños de `n`, tu cabeza estallará. Pero según el salto de fe, si suponemos que las dos llamadas recursivas (sí, puedes hacer dos llamadas recursivas) funcionan correctamente, entonces está claro que tendremos el resultado correcto sumándolas.

## 5.10. Glosario

**tipo de retorno:** La parte de la declaración de un método que indica qué tipo de valor devuelve el método.

**valor de retorno:** El valor devuelto como resultado de la llamada a un método.

**andamiaje:** Código que se utiliza durante el desarrollo del programa, pero que no forma parte de la versión final.

**lógico:** Un tipo de variable que contiene sólo los dos valores `True` y `False`.

**indicador:** Una variable (normalmente lógica) que registra una condición o información de estado.

**operador relacional:** Un operador que compara dos valores y produce un valor lógico que indica la relación entre los operandos.

**operador lógico:** Un operador que combina valores lógicos y produce valores lógicos.



## Capítulo 6

# Iteración

### 6.1. Varias asignaciones

No he hablado mucho acerca de ello, pero en Eiffel está permitido hacer más de una asignación sobre la misma variable. El efecto de la segunda asignación es el de reemplazar el valor viejo de la variable por un nuevo valor.

```
local
  pepe: INTEGER
do
  pepe := 5
  print(pepe)
  pepe := 7
  print(pepe)
end
```

La salida de este programa es 57, porque la primera vez que imprimimos **pepe** su valor es 5, y la segunda vez su valor es 7.

Esto de hacer **varias asignaciones** es el motivo por el que describí las variables como *contenedores* de valores. Cuando asignas un valor a una variable, cambias el contenido del contenedor, como se muestra en la figura:



Aunque hacer varias asignaciones a menudo es útil, debes usar esa característica con precaución. Si el valor de las variables está cambiando constantemente en diferentes partes del programa, puede que el código se haga más difícil de leer y depurar.

### 6.2. Iteración

Una de las cosas para las que son usados a menudo los ordenadores es para automatizar tareas repetitivas. Repetir tareas idénticas o similares sin cometer errores es algo que los ordenadores

hacen bien y las personas hacen regular.

Ya hemos visto programas que utilizan la recursividad para realizar repeticiones, como en `n_lineas` y `cuenta_atras`. Este tipo de repetición se llama **iteración**, y Eiffel proporciona una característica que hace más sencillo escribir programas iterativos. Esta característica es la sentencia `loop`.

### 6.3. La sentencia `loop`

Usando una sentencia `loop`, podemos volver a escribir `cuenta_atras`:

```
cuenta_atras(n: INTEGER) is
  local
    i: INTEGER
  do
    from
      i := n
    until
      i = 0
    loop
      print(i)
      print("%N")
      i := i - 1
    end
    print("¡Lanzamiento!")
  end
```

Lo que significa esta sentencia es, “Antes que nada, haz que `i` valga lo mismo que vale `n`. A partir de entonces, y hasta que `i` sea igual a cero, imprime el valor de `i` y una nueva línea, y después reduce el valor de `i` en 1. Cuando se llegue a cero, imprime la palabra ‘¡Lanzamiento!’”

De manera más formal, el flujo de ejecución para una sentencia `loop` es como sigue:

1. Se ejecutan una sola vez las sentencias situadas entre `from` y `until`.
2. Se evalúa la condición o condiciones situada entre `until` y `loop`, lo que deberá dar un valor de `True` o `False`.
3. Si todas las condiciones son verdaderas, salir de la sentencia `loop` y continuar la ejecución en la línea siguiente a la palabra `end`.
4. En caso contrario, si alguna o todas las condiciones son falsas, ejecutar cada instrucción contenida entre `loop` y `end`, y volver de nuevo al paso 2.

Este tipo de flujo se llama **bucle**. Observa que si la condición es verdadera la primera vez que se entra en el bucle, no se ejecutarán nunca las sentencias contenidas dentro del bucle (las situadas entre `loop` y `end`). Estas sentencias contenidas en el bucle a menudo se llaman el **cuerpo** del bucle.

De igual forma, las sentencias situadas entre `from` y `until`, si las hay, se llaman la **inicialización** del bucle. En la inicialización del bucle se colocan sentencias que preparan al bucle antes de comenzar su ejecución. Suele usarse, normalmente, para dar valores iniciales a las variables que forman parte de la condición de salida del bucle.

El cuerpo del bucle debería cambiar el valor de una o más variable de forma que, en un momento dado, la condición se haga verdadera y el bucle finalice. En caso contrario, el bucle se repetiría para

siempre, lo que se denomina bucle **infinito**. Una fuente inagotable de entretenimiento para los informáticos es la observación de que las instrucciones del champú, “enjabonar, aclarar, repetir”, es un bucle infinito.

En el caso de `cuenta_atras`, podemos demostrar que el bucle terminará porque sabemos que el valor de `n` es finito, y podemos ver que el valor de `n` se hace más pequeño cada vez que se atraviesa el bucle (en cada **iteración**), por lo que llegará el momento en el que se haga cero. En otros casos no es tan fácil de decir:

```

secuencia(n: INTEGER) is
  local
    i: INTEGER
  do
    from
      i := n
    until
      i = 1
    loop
      print(i)
      if i \ 2 = 0 then -- i es par
        i := i / 2
      else -- i es impar
        i := i * 3 + 1
      end
    end
  end
end

```

La condición de salida de este bucle es `i = 1`, por lo que el bucle continuará hasta que `i` sea 1, lo que hará que la condición sea verdadera.

En cada iteración, el programa imprime el valor de `i` y después comprueba si es par o impar. Si es par, el valor de `i` se divide entre dos. Si es impar, se sustituye por  $3i + 1$ . Por ejemplo, si el valor inicial (el argumento que se le pasa a `secuencia`) es 3, la secuencia resultante sería 3, 10, 5, 16, 8, 4, 2, 1.

Como `i` a veces crece y a veces decrece, no hay una prueba obvia de que `i` siempre acabará alcanzando el valor 1, o que el programa acabará. Para algunos valores particulares de `n`, podemos demostrar la detención del programa. Por ejemplo, si el valor inicial es una potencia de dos, entonces el valor de `i` será par cada vez que se atraviesa el bucle, hasta que llegue a 1. El ejemplo anterior finaliza con una secuencia así, comenzando con 16.

Valores particulares aparte, la pregunta interesante es si podemos demostrar que este programa terminará para *todos* los valores de `n`. ¡Hasta ahora, nadie ha sido capaz de demostrar la verdad o falsedad de esa afirmación!

## 6.4. Tablas

Una de las cosas para las que son buenas los bucles es la generación e impresión de datos tabulares. Por ejemplo, antes de que los ordenadores estuvieran disponibles para el gran público, la gente tenía que calcular logaritmos, senos y cosenos, y otras funciones matemáticas comunes a mano.

Para hacer la cosa más fácil, existían libros que contenían grandes tablas donde podías encontrar el valor de algunas funciones. Crear esas tablas era lento y aburrido, y el resultado tenía tendencia a estar lleno de errores.

Cuando los ordenadores aparecieron en escena, una de las reacciones iniciales fue “¡Esto es fantástico! Podemos usar los ordenadores para generar las tablas, de manera que no habrán más errores”. Esto se hizo realidad (principalmente), pero la cosa no quedó ahí. Pronto los ordenadores (y las calculadoras) fueron lo bastante potentes como para hacer que las tablas se volvieran obsoletas.

Bueno, casi. Se da el caso de que para ciertas operaciones, los ordenadores usan tablas de valores para obtener una respuesta aproximada, y luego realizan cálculos para mejorar la aproximación. En algunos casos, han existido errores en las tablas subyacentes, algunos tan famosos como la tabla usada por el Intel Pentium original para realizar divisiones reales.

Aunque una “tabla de logaritmos” no es tan útil como una vez lo fue, todavía sigue siendo un buen ejemplo de iteración. El siguiente programa imprime una secuencia de valores en la columna de la izquierda, y sus logaritmos en la columna de la derecha:

```
local
  x: DOUBLE
do
  from
    x := 1.0
  until
    x = 10.0
  loop
    print(x) print(" ") print(x.log) print("%N")
    x := x + 1.0
  end
end
```

Observa la sentencia `x := 1.0` en la sección de inicialización del bucle. Significa que, antes de que el bucle se ejecute, la variable `x` contendrá inicialmente el valor `1.0` (recuerda que la inicialización sólo se ejecuta una vez). Lo normal es que esa variable cambie su valor dentro del cuerpo del bucle hasta que alcance un valor que haga que la condición de terminación (`x = 10.0`) se cumpla, y el bucle finalice. Y efectivamente, la variable cambia su valor en cada iteración con la sentencia `x := x + 1.0`.

Una pregunta que podrías hacerte ahora es: ¿ocurre algo si coloco la asignación `x := 1.0` antes de la palabra `from`, y deajo vacía la sección de inicialización del bucle? Es decir, una cosa así:

```
local
  x: DOUBLE
do
  x := 1.0
  from
  until
    x = 10.0
  loop
    print(x) print(" ") print(x.log) print("%N")
    x := x + 1.0
  end
end
```

La respuesta es: *nada*. El programa se comportará de igual forma, puesto que la sentencia `x := 1.0` fuera del bucle se ejecutará, evidentemente, una sola vez, al igual que ocurre si la colocamos en la inicialización del bucle. Entonces, ¿para qué sirve la sección de inicialización? La respuesta aquí también es muy sencilla: para hacer más claro y legible el código del bucle. Al situar un

grupo de sentencias en la inicialización del bucle, estamos diciendo expresamente que ese código inicializa y prepara el bucle para su ejecución. Recuerda que lo importante no es sólo que el programa funcione, sino que también sea legible y fácil de depurar.

La salida de este programa es

```
1.0 0.0
2.0 0.6931471805599453
3.0 1.0986122886681098
4.0 1.3862943611198906
5.0 1.6094379124341003
6.0 1.7917594692280550
7.0 1.9459101490553132
8.0 2.0794415416798357
9.0 2.1972245773362196
```

Mirando estos valores, ¿puedes decir qué base usa por defecto la función `log`?

Ya que las potencias de dos son tan importantes en la Informática a menudo queremos calcular logaritmos en base 2. Para hacerlo, debemos usar la siguiente fórmula:

$$\log_2 x = \frac{\log_e x}{\log_e 2} \quad (6.4.1)$$

Cambiando la sentencia `print(x.log)` por

```
print(x.log / (2.0).log)
```

resulta

```
1.0 0.0
2.0 1.0
3.0 1.5849625007211563
4.0 2.0
5.0 2.321928094887362
6.0 2.584962500721156
7.0 2.807354922057604
8.0 3.0
9.0 3.1699250014423126
```

Podemos ver que 1, 2, 4 y 8 son potencias de dos, porque sus logaritmos en base 2 son números redondos. Si queremos encontrar los logaritmos de otras potencias de dos, podríamos modificar el programa así:

```
local
  x: DOUBLE
do
  from
    x := 1.0
  until
    x = 100.0
  loop
    print(x) print(" ")
    print(x.log / (2.0).log) print("%N")
    x := x * 2.0
  end
end
```

Ahora, en lugar de sumar cosas a  $x$  cada vez que atravesamos el bucle, lo que produce una secuencia aritmética, multiplicamos  $x$  por algo, produciendo una secuencia **geométrica**. El resultado es:

```
1.0  0.0
2.0  1.0
4.0  2.0
8.0  3.0
16.0 4.0
32.0 5.0
64.0 6.0
```

Las tablas de logaritmos puede que no sean útiles nunca más, pero para los informáticos, ¿conocer las potencias de dos sí lo es! Alguna vez cuando tengas un momento libre, deberías memorizar las potencias de dos hasta 65536 (esto es  $2^{16}$ ).

## 6.5. Tablas bidimensionales

Una tabla bidimensional es una tabla en donde eliges una fila y una columna y lees el valor situado en la intersección. Una tabla de multiplicar es un buen ejemplo. Digamos que quieres imprimir la tabla de multiplicar de todos los números entre 1 y 6.

Una buena manera de comenzar es escribir un bucle simple que imprima los múltiplos de dos, todos en una línea.

```
local
  i: INTEGER
do
  from
    i := 1
  until
    i > 6
  loop
    print(2*i) print("  ")
    i := i + 1
  end
  print("%N")
end
```

En la inicialización del bucle, se le asigna un valor inicial a una variable llamada  $i$ , que actuará como **contador** (una de las utilidades más comunes de la sección de inicialización de los bucles es la de dar valores iniciales a los contadores). A medida que el bucle se ejecuta, el valor de  $i$  se incrementa desde 1 hasta 6, y después, cuando  $i$  se hace 7, el bucle finaliza. Cada vez que se atraviesa el bucle, imprimimos el valor  $2*i$  seguido de tres espacios en blanco. Todo esto aparece en la misma línea.

La salida de este programa es:

```
2  4  6  8  10 12
```

No está mal. El siguiente paso es **encapsular** y **generalizar**.

## 6.6. Encapsulación y generalización

Encapsular normalmente significa tomar una porción de código y envolverlo dentro de un método, permitiéndote aprovechar todas las cosas por las que son buenos los métodos. Hemos visto dos ejemplos de encapsulación, cuando escribimos `imprimir_paridad` en la sección 4.3 y `es_un_solo_digito` en la sección 5.6.

Generalizar significa coger algo específico, como imprimir múltiplos de 2, y hacerlo más general, como imprimir los múltiplos de cualquier entero.

Aquí tenemos un método que encapsula el bucle de la sección anterior y lo generaliza para imprimir múltiplos de `n`.

```
imprimir_multiplos(n: INTEGER) is
  local
    i: INTEGER
  do
    from
      i := 1
    until
      i > 6
    loop
      print(n * i) print(" ")
      i := i + 1
    end
    print("%N")
  end
```

Para encapsular, todo lo que tuve que hacer fue añadir la primera línea, que declara el nombre, parámetro y tipo de retorno (ninguno, en este caso) del método. Para generalizar, todo lo que tuve que hacer fue sustituir el valor 2 por el parámetro `n`.

Si llamo a este método con el argumento 2, obtengo la misma salida que antes. Con el argumento 3, la salida es:

```
3  6  9  12  15  18
```

y con el argumento 4, la salida es

```
4  8  12  16  20  24
```

Ahora probablemente ya te puedes imaginar cómo vamos a imprimir la tabla de multiplicar: llamaremos repetidamente a `imprimir_multiplos` con diferentes argumentos. De hecho, vamos a usar otro bucle para iterar a través de las filas.

```
local
  i: INTEGER
do
  from
    i := 1
  until
    i > 6
  loop
    imprimir_multiplos(i)
    i := i + 1
  end
end
```

Antes que nada, observa lo parecido que es este bucle con el que está dentro de `imprimir_multiplos`. Todo lo que hice fue sustituir la sentencia de impresión por la llamada a un método.

La salida de este programa es

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36

que es una (algo fea) tabla de multiplicar. Si te molesta lo fea que queda, Eiffel te ofrece métodos que te dan más control sobre el formato de la salida, pero no entraré ahora en eso.

## 6.7. Métodos

En la sección anterior hablé de “todas las cosas por las que son buenos los métodos”. En este momento, podrías estar preguntándote cuáles son exactamente esas cosas. Aquí tenemos algunas de las razones por las que los métodos son útiles:

- Al darle un nombre a una secuencia de sentencias, haces que tu programa sea más fácil de leer y depurar.
- Dividir un programa grande en métodos te permite separar partes del programa, depurarlas por separado, y luego juntarlas formando un todo.
- Los métodos facilitan la recursividad y la iteración.
- Los métodos bien diseñados a menudo son útiles para muchos programas. Una vez que escribes y depuras uno, lo puedes reutilizar.

## 6.8. Más encapsulación

Para demostrar de nuevo la encapsulación, cogeré el código de la sección anterior y lo envolveré dentro de un método:

```
imprimir_tabla_de_multiplicar is
  local
    i: INTEGER
  do
    from
      i := 1
    until
      i > 6
    loop
      imprimir_multiplos(i)
      i := i + 1
    end
  end
end
```



El proceso que estoy mostrando es un típico plan de desarrollo. Desarrollas código gradualmente añadiendo líneas a `make` o en cualquier otro lugar, y luego cuando lo tienes funcionando, lo extraes y lo envuelves en un método.

La razón por la que esto es útil es que a veces, cuando estas empezando a escribir, no sabes exactamente cómo dividir el programa en métodos. Esta aproximación te permite diseñar conforme vas avanzando.

## 6.9. Variables locales

En estos momentos, te podrías estar preguntando cómo podemos usar la misma variable `i` en dos sitios distintos: `imprimir_multiplos` e `imprimir_tabla_de_multiplicar`. ¿No dije que sólo puedes declarar una variable una sola vez? ¿Y ésto no causa problemas cuando uno de los métodos cambia el valor de la variable?

La respuesta a ambas preguntas es “no”, porque la `i` en `imprimir_multiplos` y la `i` en `imprimir_tabla_de_multiplicar` *no son la misma variable*. Tienen el mismo nombre, pero no se refieren al mismo espacio de almacenamiento, y cambiar el valor de una no tiene efecto sobre la otra.

Las variables declaradas dentro de una definición de método se llaman **variables locales** (es por eso por lo que se declaran dentro de la sección que empieza con la palabra clave `local`). No puedes acceder a una variable local desde fuera de su método “hogar”, y eres libre de tener varias variables con el mismo nombre, siempre y cuando no estén declaradas dentro del mismo método.

A menudo es una buena idea usar diferentes nombres de variable en métodos diferentes, para evitar confusión, pero hay buenas razones para reutilizar nombres. Por ejemplo, es común usar los nombres `i`, `j` y `k` como contadores de bucles. Si evitas usarlas en un método simplemente porque ya las has usado en otra parte, probablemente harás que tu programa sea más difícil de leer.

## 6.10. Más generalización

Como otro ejemplo de generalización, imagínate que quieres un programa que pueda imprimir una tabla de multiplicar de cualquier tamaño, no solo de 6x6. Podrías añadir un parámetro a `imprimir_tabla_de_multiplicar`:

```
imprimir_tabla_de_multiplicar(altura: INTEGER) is
  local
    i: INTEGER
  do
    from
      i := 1
    until
      i > altura
    loop
      imprimir_multiplos(i)
      i := i + 1
    end
  end
```

Reemplacé el valor 6 por el parámetro `altura`. Si invoco a `imprimir_tabla_de_multiplicar` con el argumento 7, obtengo

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36
7	14	21	28	35	42

lo que está bien, excepto que probablemente quiera que la tabla sea cuadrada (que tenga el mismo número de filas y columnas), lo que significa que tengo que añadir otro parámetro a `imprimir_multiplos`, para especificar cuántas columnas debería tener la tabla.

Sólo para molestar, también llamaré a este parámetro **altura**, demostrando así que métodos diferentes pueden tener parámetros con el mismo nombre (justamente igual que lo que ocurre con las variables locales):

```
imprimir_multiplos(n, altura: INTEGER) is
  local
    i: INTEGER
  do
    from
      i := 1
    until
      i > altura
    loop
      print(n * i) print(" ")
      i := i + 1
    end
    nueva_linea
  end

imprimir_tabla_de_multiplicar(altura: INTEGER) is
  local
    i: INTEGER
  do
    from
      i := 1
    until
      i > altura
    loop
      imprimir_multiplos(i, altura)
      i := i + 1
    end
  end
```

Observa que cuando he añadido un nuevo parámetro, he cambiado la primera línea del método (la interfaz o prototipo), y además he cambiado el lugar en el que el método es invocado en `imprimir_tabla_de_multiplicar`. Como se espera, este programa genera una tabla cuadrada de 7x7:

1	2	3	4	5	6	7
2	4	6	8	10	12	14
3	6	9	12	15	18	21
4	8	12	16	20	24	28

5	10	15	20	25	30	35
6	12	18	24	30	36	42
7	14	21	28	35	42	49

Cuando generalizas un método adecuadamente, a menudo encuentras que el programa resultante tiene capacidades que no pretendías que tuviera. Por ejemplo, puede que te hayas dado cuenta que la tabla de multiplicar es simétrica, porque  $ab = ba$ , así que todas las entradas en la tabla aparecen dos veces. Podrías ahorrar tinta si imprimes sólo la mitad de la tabla. Para hacer eso, sólo tienes que cambiar una línea de `imprimir_tabla_de_multiplicar`. Cambia

```
imprimir_multiplos(i, altura)
```

por

```
imprimir_multiplos(i, i)
```

y obtendrás

1						
2	4					
3	6	9				
4	8	12	16			
5	10	15	20	25		
6	12	18	24	30	36	
7	14	21	28	35	42	49

Te dejaré que pienses el por qué y cómo funciona esto.

## 6.11. Glosario

**bucle:** Una sentencia que se ejecuta repetidamente hasta que se satisface una condición.

**bucle infinito:** Un bucle cuya condición de parada es siempre falsa.

**cuerpo:** Las sentencias dentro del bucle.

**inicialización:** Las sentencias que se ejecutan una sola vez al comienzo del bucle.

**iteración:** Un paso a través de (la ejecución del) cuerpo del bucle, incluyendo la evaluación de la condición.

**encapsular:** Dividir un programa grande y complejo en componentes (como los métodos) y aislar los componentes unos de otros (por ejemplo, usando variables locales).

**variable local:** Una variable que se declara dentro de un método y que existe sólo dentro de ese método. No se puede acceder a las variables locales desde fuera del método en las que están declaradas, y no interfieren con otros métodos.

**generalizar:** Sustituir algo innecesariamente específico (como un valor constante) por algo apropiadamente general (como una variable o un parámetro). La generalización hace al código más versátil, le otorga más probabilidad de ser reutilizado, y a veces incluso más fácil de escribir.

**plan de desarrollo:** Un proceso de desarrollo de un programa. En este capítulo, he mostrado un estilo de desarrollo basado en desarrollar código para que haga cosas sencillas y específicas, y luego encapsularlo y generalizarlo. En la sección 5.2 mostré una técnica que llamé desarrollo incremental. En próximos capítulos sugeriré otros estilos de desarrollo.

## Capítulo 7

# Cadenas y cosas

### 7.1. Invocando métodos sobre objetos

En capítulos anteriores hemos visto que la mayoría de los métodos predefinidos en Eiffel (a excepción del método `print`) actúan *sobre* un valor, y la sintaxis para ello es indicar el valor, seguido de un punto y el nombre del método, como por ejemplo:

```
local
  x, y: DOUBLE
do
  x := 2.4
  y := x.cos
end
```

En este caso, el método `cos` actúa *sobre* el objeto contenido en la variable `x` (el valor 2.4).

Hasta ahora no he proporcionado una definición de **objeto**, y todavía no puedo proporcionar una definición completa, pero es el momento de intentarlo.

En Eiffel y otros lenguajes orientados a objetos, los objetos son colecciones de datos relacionados que vienen junto con un conjunto de métodos. Esos métodos operan sobre los objetos, realizando cálculos y a veces modificando los datos del objeto.

Por ejemplo, en Eiffel el número 2.4 es un objeto de tipo `DOUBLE`. Los objetos de este tipo poseen un conjunto de métodos asociados, que son aquellos métodos que actúan sobre objetos de tipo `DOUBLE`. Algunos de esos métodos son `cos`, `tan`, `log` o `rounded`. Es por eso por lo que, cuando hacemos `(2.4).cos`, en realidad estamos pidiéndole al objeto 2.4 que ejecute su método `cos` sobre él.

Hasta ahora hemos visto objetos de muy diferentes tipos: enteros, reales, cadenas, etcétera. Centrémonos en las cadenas. Las cadenas son objetos de tipo `STRING`. Basándonos en la definición de objeto, podríamos preguntarnos “¿Cuáles son los datos contenidos en un objeto `STRING`?” y “¿Cuáles son los métodos que podemos invocar sobre objetos `STRING`?”.

Los datos que contiene un objeto `STRING` son las letras de la cadena. Existen muchos métodos que trabajan sobre `STRINGs`, pero sólo usaré unos cuantos en este libro. El resto lo puedes encontrar en la documentación de Eiffel.

El primer método que veremos es `item`, que permite extraer letras de una cadena. Con idea de almacenar el resultado, necesitamos un tipo de variable que permita almacenar letras individuales (a diferencia de cadenas completas). A las letras individuales se las llama caracteres, y el tipo de variable que las almacena se llama `CHARACTER`.

Los `CHARACTERs` funcionan igual que otros tipos que hemos visto:

```
local
  pepe: CHARACTER
do
  pepe := 'c'
  if pepe = 'c' then
    print(pepe)
  end
end
```

Los caracteres aparecen entre comillas simples ('c'). A diferencia de las cadenas (que aparecen entre comillas dobles), un valor de tipo carácter sólo puede contener una sola letra entre las comillas simples.

Así es como se usa el método `item`:

```
local
  fruta: STRING
  letra: CHARACTER
do
  fruta := "banana"
  letra := fruta.item(1)
  print(letra)
end
```

La sintaxis `fruta.item` significa que estoy invocando el método `item` sobre el objeto llamado `fruta`. Estoy pasando al método el argumento 1, que indica que me gustaría saber cuál es la primera letra de la cadena. El resultado es un carácter, que se almacena en un `CHARACTER` llamado `letra`. El resultado es:

```
b
```

como cabría esperar, puesto que la cadena `"banana"` comienza con la letra `b`. La segunda letra es `a` y la tercera es `n`.

## 7.2. Longitud

El segundo método de `STRING` que vamos a ver es `count`, que devuelve el número de caracteres que contiene la cadena. Por ejemplo:

```
local
  longitud: INTEGER
do
  longitud := fruta.count
end
```

`count` no recibe ningún argumento, y devuelve un entero, que en este caso es 6 (porque la cadena `"banana"` tiene 6 caracteres).

Para obtener la última letra de una cadena, podemos hacer algo así:

```
local
  longitud: INTEGER
  ultimo: CHARACTER
```

```

do
    longitud := fruta.count
    ultimo := fruta.item(longitud)
end

```

### 7.3. Recorrido

Algo que es común hacer con una cadena es empezar por el principio, coger el carácter de turno, hacer algo con él, y continuar hasta el final. Este patrón de procesamiento se denomina **recorrido**. Una manera natural de codificar un recorrido es con una sentencia `loop`:

```

local
    indice: INTEGER
    letra: CHARACTER
do
    from
        indice := 1
    until
        indice > fruta.count
    loop
        letra := fruta.item(indice)
        print(letra) print("%N")
        indice := indice + 1
    end
end

```

Este bucle recorre la cadena e imprime cada letra en una línea aparte. Observa que la condición es `indice > fruta.count`, lo que significa que el bucle terminará cuando el índice se salga de los límites de la cadena. El último carácter al que accedemos es aquel que tiene el índice `fruta.count`.

El nombre de la variable contador es `indice`. Un **índice** es una variable o un valor usado para especificar un miembro concreto de un conjunto ordenado (en este caso el conjunto de los caracteres de la cadena). El índice indica (de ahí el nombre) cuál es el que quieres. El conjunto tiene que estar ordenado para que cada letra tenga su índice y cada índice se refiera a un único carácter.

Como ejercicio, escribe un método que reciba una cadena como argumento e imprima sus letras al revés, cada una en una línea.

### 7.4. Errores en tiempo de ejecución

En la sección 1.3.2 hablé acerca de los errores en tiempo de ejecución, que son errores que no aparecen hasta que el programa ha empezado a ejecutarse. En Eiffel, los errores en tiempo de ejecución se denominan **excepciones**.

Hasta ahora, probablemente no hayas visto muchos errores en tiempo de ejecución, porque no hemos hecho muchas cosas que puedan causarlos. Bueno, pues ahora lo haremos. Si usas el método `item` y le suministras un índice negativo o mayor que `count`, obtendrás un mensaje de error. Concretamente, un `"Require assertion violated: valid_index"`. De él, podemos deducir que se ha violado una regla necesaria para usar el método `item`, y esa regla es que debemos suministrar al método `item` un índice válido. Inténtalo y verás.

Cada vez que el programa causa una excepción, imprime un mensaje de error indicando el tipo de error y en qué lugar del programa se originó. Después el programa finaliza.

## 7.5. Leer la documentación

Si miras en la documentación de Eiffel el fichero `STRING.html`, podrás ver la siguiente documentación (o algo parecido a ésto):

```
item(i: INTEGER): CHARACTER
  -- Character at position i.
  require
    valid_index: valid_index(i)
```

La primera línea es la **interfaz** del método, la cual indica el nombre del método, el tipo de sus parámetros, y el tipo de retorno.

A continuación viene un comentario que expresa lo que hace el método. En este caso, la explicación es muy clara y escueta: “**Character at position i.**”, lo que significa que devuelve el carácter situado en la posición *i*, donde *i* es el nombre del parámetro.

Tras el comentario viene la palabra clave **require**. Esa palabra es el comienzo de una sección de texto donde se recogen las condiciones que se deben cumplir para que el método pueda hacer bien su trabajo (de ahí el nombre **require**, pues describe qué se requiere para que el método funcione). La única condición que aparece viene etiquetada con **valid\_index**, (justamente el nombre que aparece en el mensaje de error cuando le pasamos a **item** un índice erróneo), y la condición en sí misma es **valid\_index(i)**.

¿Qué representa esa condición? Miramos un poco más abajo en la documentación y encontramos:

```
valid_index(i: INTEGER): BOOLEAN
  -- True when i is valid (i.e., inside actual bounds).
  ensure
    definition: Result = (1 <= i and i <= count)
```

Es decir, y resumiendo: Para que **item** pueda funcionar correctamente, se debe cumplir la condición **valid\_index(i)**, y ésta condición se define como **(1 <= i and i <= count)**. Por tanto, deducimos que para que **item** funcione, su parámetro *i* debe estar comprendido entre 1 y **count**.

La documentación expresa, claramente y sin lugar a interpretaciones erróneas, qué es lo que hace el método, cuál es su interfaz, y qué condiciones se deben cumplir si queremos que el método funcione. En consecuencia, nos dice todo lo que necesitamos saber para poder usar el método. Y lo mejor de todo es que ni siquiera nos ha hecho falta mirar el código interno del método.

A este formato especial de documentación se le denomina **forma contractual**, **forma corta**, o **contrato** del método.

## 7.6. El método `first_index_of`

En cierta forma, **first\_index\_of** es el opuesto a **item**. **item** recibe un índice y devuelve el carácter situado en ese índice. **first\_index\_of** recibe un carácter y devuelve el índice en el que se encuentra ese carácter por primera vez dentro de la cadena.

**item** fracasa si el índice está fuera del rango, y causa una excepción. **first\_index\_of** fracasa si el carácter no aparece en la cadena, y devuelve el valor 0.

```
local
  fruta: STRING
```

```

    indice: INTEGER
do
    fruta := "banana"
    indice := fruta.first_index_of('a')
end

```

Esto busca el índice de la letra 'a' en la cadena. En este caso, aunque la letra aparezca tres veces, `first_index_of` devuelve el índice de la primera aparición del carácter.

Si queremos buscar posteriores apariciones, podemos usar el método `index_of`. `index_of` es un método semejante a `first_index_of`, pero en su lugar recibe un segundo argumento que indica en qué posición de la cadena se empieza a buscar. Si invocamos:

```
indice := fruta.index_of('a', 3)
```

comenzará a buscar en la tercera letra y encontrará la segunda a, que está en el índice 4. Si la letra resulta que está en el índice inicial, la respuesta será el índice inicial. Así,

```
indice := fruta.index_of('a', 4)
```

devuelve 4. Si miramos en la documentación, veremos qué es lo que ocurriría si el índice inicial estuviese fuera de rango:

```

index_of(c: CHARACTER; start_index: INTEGER): INTEGER
-- Index of first occurrence of c at or after start_index,
-- 0 if none.
require
    valid_start_index: start_index >= 1 and start_index <= count + 1
ensure
    Result /= 0 implies item(Result) = c

```

Como se observa, la condición que aparece en la cláusula **require** es que el índice inicial debe estar obligatoriamente entre 1 y `count + 1`, ambos inclusive. Por tanto, si proporcionamos uno que no esté dentro de ese rango, se generará una excepción por no cumplirse la condición de **require**.

## 7.7. Diseño por Contrato

De la sección anterior se desprende que una de las informaciones más útiles que podemos extraer de la documentación de un método es su cláusula **require**. Esta cláusula establece las condiciones que se han de dar justo antes de llamar a un método para que éste realice bien su trabajo. Cuando esta condición no se cumple y nosotros nos obstinamos en pedirle al método que se ejecute, se genera una excepción, y en el mensaje de error correspondiente aparece la etiqueta de la condición **require** que no se ha cumplido (como ocurrió antes con `valid_index`).

Además de la cláusula **require**, al mirar la documentación del método `index_of` observamos que aparece una cláusula adicional llamada **ensure**. Esta cláusula representa las condiciones que se cumplirán tras finalizar la ejecución del método correspondiente. En el caso de `index_of` sólo aparece la condición `Result /= 0 implies item(Result) = c`, que podría leerse como “si el valor de retorno del método es distinto de cero, entonces el carácter recibido como parámetro se encontrará justamente en la posición indicada por ese valor de retorno”. Como se puede apreciar, tal y como ya apuntábamos en el capítulo 1, la expresión formal es mucho más concisa y menos ambigua que la descripción en lenguaje natural.



Pero aún hay más. Este estilo de documentación aporta un punto de vista diferente a la hora de utilizar métodos ajenos: que el método llamante y el método llamado son partes contrarias que establecen su relación mediante un **contrato** que refleja las condiciones de utilización de un método por parte de otro. Ese contrato establece las obligaciones de una y otra parte en la relación establecida (la relación no es más que el deseo por parte del método llamante —el cliente— de hacer uso de los servicios ofrecidos por el método llamado —el proveedor—).

El contrato queda formalizado con las cláusulas **require** y **ensure** del método llamado, que es el proveedor del servicio. De alguna manera, la cláusula **require** establece las obligaciones que el llamante debe cumplir si desea hacer uso del método llamado. El método llamado impone la obligación de que el llamante cumpla las condiciones que aparezcan en **require**. Por otra parte, la cláusula **ensure** representa lo que el método llamado garantiza que pasará si realiza su trabajo, y por consiguiente es una obligación para el método llamado.

A las condiciones incluidas en la cláusula **require**, se las llama globalmente **precondición**. Y a las incluidas en la cláusula **ensure**, se las denomina globalmente **postcondición**.

Para clarificar un poco más las cosas, las resumiré en el siguiente cuadro:

	Obligaciones
Cliente (método llamante)	Satisfacer la precondición
Proveedor (método llamado)	Satisfacer la postcondición

Este estilo de programación, en el cual cada método queda complementado con su precondición y postcondición, se denomina **Diseño por Contrato**.

Lo interesante del caso es que ambas cláusulas, **require** y **ensure**, se pueden adjuntar al código fuente del método, formando parte indivisible del mismo, de tal manera que en tiempo de ejecución se comprueben sistemáticamente las condiciones contractuales para detectar errores que de otra manera quedarían oscuros y difíciles de encontrar.

Hablaremos más de todo esto en posteriores capítulos.

## 7.8. Iterar y contar

El siguiente programa cuenta el número de veces que aparece la letra 'a' en una cadena:

```

local
  fruta: STRING
  longitud, contador, indice: INTEGER
do
  fruta := "banana"
  longitud := fruta.count
  from
    contador := 0
    indice := 1
  until
    indice > longitud
  loop
    if fruta.item(indice) = 'a' then
      contador := contador + 1
    end
    indice := indice + 1
  end
  print(contador)
end

```

Este programa utiliza una variable **contador**. La variable **contador** se inicializa a cero y luego se incrementa cada vez que encontramos una 'a' (**incrementar** es sumar uno; es lo opuesto a **decrementar**). Cuando salimos del bucle, **contador** contiene el resultado: el número total de "as".

Como ejercicio, encapsula este código en un método llamado **cuenta\_letras**, y generalízalo de manera que acepte la cadena y la letra como argumentos.

Como segundo ejercicio, reescribe el método de forma que utilice **index\_of** para localizar las "as", en lugar de comprobar los caracteres uno a uno.

## 7.9. Caracteres y números

Los caracteres y los enteros son cosas distintas. Por ese motivo, el carácter '3' no representa al número entero 3, sino al carácter cuyo símbolo es el dígito 3.

A veces nos interesa tratar a los caracteres como dígitos. Por ejemplo, puede ser conveniente determinar a qué número entero representa un cierto carácter, como en el caso anterior. Para ello, los caracteres disponen de un método llamado **value**, que aplicado sobre un carácter como '3', devuelve el número correspondiente. Por ejemplo:

```
local
  letra: CHARACTER
  x: INTEGER
do
  letra := '6'
  x := letra.value
end
```

convierte **letra** en el correspondiente dígito, interpretándolo como un número en base 10.

Por otra parte, el método **code**, aplicado sobre un carácter, devuelve un entero que corresponde al código de ese carácter dentro del conjunto de caracteres de la máquina. El más usado de todos los conjuntos de caracteres es el ASCII, en el cual todo carácter se representa como un número de 7 bits, lo que da un total de 128 caracteres diferentes<sup>1</sup>.

¡Cuidado! No confundas el código de un carácter con el número que podría representar ese carácter. Siguiendo el ejemplo del carácter '6', su valor como número sería 6, pero su código en el conjunto ASCII es 54.

El código de un carácter puede usarse para iterar a través de las letras del alfabeto en orden. Por ejemplo, en el libro de Robert McCloskey *Make Way for Ducklings*, los nombres de los patos formaban una serie alfabética: Jack, Kack, Lack, Mack, Nack, Ouack, Pack and Quack.

He aquí un bucle que imprime esos nombres en orden:

```
local
  letra: CHARACTER
do
  from
    letra := 'J'
  until
    letra > 'Q'
  loop
```

---

<sup>1</sup>El llamado ASCII extendido representa cada carácter con un número de 8 bits, lo que supone un total de 256 caracteres, incluyendo algunos tan esenciales para los hispanohablantes como la ñe o las vocales acentuadas.

```

        print(letra) print("ack%N")
        letra := (letra.code + 1).to_character
    end
end

```

Observa que para obtener la siguiente letra, tomamos el código de la actual, le sumamos uno, y el resultado lo convertimos en carácter usando el método `to_character`, que se aplica sobre enteros.

Una pregunta que surge inmediatamente es: ¿no hay otra manera de hacer el bucle que no sea trabajando directamente sobre el código del carácter? Efectivamente, la hay. Existe un método llamado `next` que, aplicado sobre un carácter, devuelve el carácter siguiente en el conjunto de caracteres de la máquina. Es decir, `'b'.next` devuelve `'c'`. El programa modificado de esta manera queda:

```

local
  letra: CHARACTER
do
  from
    letra := 'J'
  until
    letra > 'Q'
  loop
    print(letra) print("ack%N")
    letra := letra.next
  end
end

```

Bastante más claro y natural. La salida de este programa es:

```

Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack

```

Por supuesto, esto no está del todo bien porque nos hemos equivocado en “Ouack” y “Quack”. Como ejercicio, modifica el programa para corregir este error.

## 7.10. Las cadenas cambian

Si echas un vistazo a la documentación de los métodos de `STRING`, verás a `to_upper` y `to_lower`. Esos métodos *cambian* la cadena sobre la que se aplican, poniendo todos sus caracteres en mayúsculas y minúsculas, respectivamente. Por ejemplo:

```

local
  nombre: STRING
do
  nombre := "Alan Turing"
  nombre.to_upper
end

```

Después de que se haya ejecutado la última línea, la variable `nombre` contendrá el valor "ALAN TURING".

### 7.11. Las cadenas son incomparables

A menudo es necesario comparar cadenas para ver si son la misma, o ver cuál viene primero en orden alfabético. Para esto último podemos usar los operadores `<`, `<=`, `>` o `>=`, pero para comprobar la igualdad entre cadenas no es posible usar el operador `=`.

Para poder ver si dos cadenas son iguales, debemos usar el método `is_equal`. Por ejemplo:

```
local
  nombre1, nombre2: STRING
  indicador: INTEGER
do
  nombre1 := "Alan Turing"
  nombre2 := "Ada Lovelace"
  if nombre1.is_equal(nombre2) then
    print("Los nombres son iguales.%N")
  end
end
```

Aunque podemos comprobar si, por ejemplo, `nombre1 <= nombre2`, o `nombre1 > nombre2`, vamos a ver un método que unifica todas las posibles comparaciones en una sola: la llamada `nombre1.compare(nombre2)` devuelve:

- 0 si las dos cadenas son iguales
- -1 si `nombre1 < nombre2`
- 1 si `nombre1 > nombre`

Como ejemplo, tenemos:

```
local
  nombre1, nombre2: STRING
  indicador: INTEGER
do
  indicador := nombre1.compare(nombre2)
  if indicador = 0 then
    print("Los nombres son iguales.%N")
  elseif indicador < 0 then
    print("nombre1 va antes que nombre2.%N")
  else -- indicador > 0
    print("nombre2 va antes que nombre1.%N")
  end
end
```

La sintaxis es un poco extraña. Para comparar dos cadenas, tenemos que aplicar el método sobre una de ellas y pasar la otra como un argumento.

El valor de retorno de `is_equal` está bastante claro; `True` si las dos cadenas contienen los mismos caracteres, y `False` en caso contrario.

Por completar el tema, debo admitir que está permitido, pero normalmente será incorrecto, usar el operador `=` con cadenas. Pero lo que eso significa no tendrá sentido hasta más tarde, así que por ahora, no lo hagas.

## 7.12. Glosario

**objeto:** Una colección de datos relacionados que vienen con un conjunto de métodos que operan sobre ellos. Hasta ahora hemos usado objetos enteros, reales, cadenas, lógicos y caracteres.

**índice:** Una variable o valor usado para seleccionar uno de los miembros de un conjunto ordenado, como un carácter de una cadena.

**recorrer:** Iterar a través de los elementos de un conjunto realizando una operación similar con cada uno de ellos.

**contador:** Una variable usada para contar algo, normalmente inicializada a cero e incrementada posteriormente.

**incrementar:** Sumar uno al valor de una variable.

**decrementar:** Restar uno al valor de una variable.

**excepción:** Un error en tiempo de ejecución. Las excepciones causan que finalice la ejecución de un programa.

## Capítulo 8

# Objetos interesantes

### 8.1. ¿Qué es interesante?

Aunque las cadenas son objetos, no son objetos muy interesantes, porque

- No tienen variables de instancia.
- No tienes que usar una sentencia `create` para crear una.

En este capítulo, vamos a usar dos nuevos tipos de objetos, `Punto` y `Rectangulo`, para lo cual será necesario crear dos ficheros, uno por cada uno de los dos nuevos tipos que hay que definir. Posteriormente, en nuestros programas usaremos objetos de esos tipos, y el compilador sabrá manejarlos convenientemente al disponer de las definiciones que nosotros previamente hemos elaborado.

Cada tipo se define en una clase independiente. La definición de cada clase debe guardarse en un fichero distinto, cuyo nombre debe ser el nombre (en minúsculas) de la clase, y con la extensión “.e”, de Eiffel. Así que tendremos dos nuevos ficheros fuente: `punto.e` y `rectangulo.e`. El contenido del fichero `punto.e` debe ser:

```
class PUNTO

creation make

feature
  x, y: DOUBLE
  make(xp, yp: DOUBLE) is
    -- Inicializar un objeto punto
    do
      x := xp
      y := yp
    end
end
```

Y el de `rectangulo.e` es:

```
class RECTANGULO

creation make
```

```

feature
  x, y, alto, ancho: DOUBLE

  make(xr, yr, alr, anr: DOUBLE) is
    -- Inicializar un objeto rectángulo
    do
      x := xr
      y := yr
      alto := alr
      ancho := anr
    end
end

```

Antes de empezar, quiero dejar claro que esos puntos y rectángulos no son objetos gráficos que aparecen en la pantalla. Son variables que contienen datos, justamente igual que los enteros y los reales. Como cualquier otra variable, serán utilizadas internamente para realizar cálculos.

## 8.2. Objetos PUNTO

Al nivel más básico, un punto está formado por dos números (coordenadas) que pueden ser tratados colectivamente como un único objeto. En notación matemática, los puntos a menudo se escriben entre paréntesis, con una coma separando las coordenadas. Por ejemplo,  $(0, 0)$  indica el origen, y  $(x, y)$  indica el punto situado  $x$  unidades a la derecha e  $y$  unidades hacia arriba del origen.

Para crear un nuevo punto tienes que usar la sentencia **create**, combinada con la llamada al método **make** del punto:

```

local
  blanco: PUNTO
do
  create blanco.make(3, 4)
end

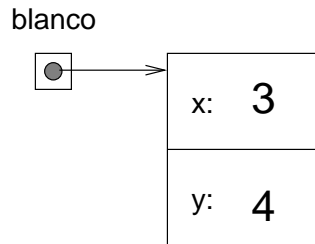
```

En la sección **local** declaramos que **blanco** es una variable de tipo **PUNTO**. La sentencia de la sección **do** representa la orden de creación del objeto. Consta de la palabra reservada **create**, seguida de la variable que va a contener el nuevo objeto, y a continuación la llamada al método **make** con los argumentos 3 y 4. Probablemente no te sorprenda saber que los argumentos son las coordenadas del nuevo punto,  $(3, 4)$ .

La ejecución de la sentencia de creación del objeto se ejecuta en dos pasos:

1. **create** construye en memoria un nuevo objeto y se lo asigna a la variable **blanco**. Ese objeto será del mismo tipo que el tipo de la variable.
2. A continuación, sobre ese objeto recién creado se aplica el método **make**, que como podemos ver en su definición (en el apartado anterior), lo que hace es guardar sus parámetros, respectivamente, en **x** e **y**, que son los datos internos del objeto; en este caso, las coordenadas del punto. Como nota de curiosidad, en una sentencia **create** sólo se pueden usar métodos que, como **make**, aparezcan declarados en la sección **creation** de la clase.

El resultado final es una **referencia** al nuevo punto. Explicaré las referencias más adelante; por ahora, lo más importante es que la variable **blanco** contiene una referencia al objeto recién creado. Existe una forma estándar de dibujar esta sentencia, como se muestra en la figura.



Como es usual, el nombre de la variable `blanco` aparece fuera de la caja, y su valor aparece dentro de la caja. En este caso, ese valor es una referencia, la cual se muestra gráficamente con un punto gordo y una flecha. La flecha apunta al objeto al que está haciendo referencia.

La caja grande muestra el objeto recién creado con sus dos valores dentro. Los nombres `x` e `y` son los nombres de los **atributos** (los datos internos del objeto).

Juntos, todas las variables, valores y objetos de un programa se denominan el **estado** del programa. Los diagramas como éste que muestran el estado de un programa se llaman **diagramas de estado**. Durante la ejecución del programa, el estado cambia, por lo que debes pensar en un diagrama de estado como en una instantánea de un momento particular en la ejecución.

### 8.3. Atributos

Las porciones de datos que forman un objeto se denominan **atributos**. Aunque todos los objetos de un mismo tipo poseen los mismos atributos, cada objeto particular posee una copia distinta e independiente de esos atributos. Cuando un objeto posee un cierto tipo, también se dice que dicho objeto es una **instancia** de ese tipo.

Es como la guantera de un coche. Cada coche es una instancia del tipo “coche”, y cada coche tiene su propia guantera. Si me pides que coja algo de la guantera de tu coche, debes decirme qué coche es el tuyo.

De igual forma, si quieres que lea el valor de un atributo, debes especificar el objeto del que quieres leer el atributo. En Eiffel eso se hace usando la “notación punto”, ya utilizada por nosotros para indicar sobre qué objeto se aplica un cierto método.

```
local
  x: DOUBLE
do
  x := blanco.x
end
```

La expresión `blanco.x` significa “ve al objeto al que se refiere la variable `blanco`, y recoge el valor de `x`”. En este caso, asignamos ese valor a la variable local llamada `x`. Observa que no existe conflicto entre la variable local llamada `x` y el atributo llamado `x`. El propósito de la notación punto es identificar a *qué* variable estás refiriéndote sin ambigüedad.

Puedes usar la notación punto como parte de cualquier expresión Eiffel, por lo que lo siguiente está permitido.

```
local
  distancia: DOUBLE
do
  print(blanco.x) print(", ") print(blanco.y)
  distancia := blanco.x * blanco.x + blanco.y * blanco.y
end
```

La primera línea imprime `3, 4`; la segunda calcula el valor `25.0`.



## 8.4. El Principio del Acceso Uniforme

Habrás observado que hemos utilizado la misma notación para acceder a los atributos de un objeto que para invocar a un método sobre el objeto: la notación punto. Es decir, indicamos el objeto, seguido de un punto, y el nombre del atributo o el método. Por ejemplo, si queremos acceder al atributo `x` del objeto `blanco`, escribimos `blanco.x`; y si queremos calcular el coseno de un número almacenado en `n`, escribiríamos `n.cos`.

Ahora imagina que, en medio de un programa, te encuentras la expresión `total := persona.salario`, en la que se almacena el salario de una persona en la variable `total`. La pregunta que puede rondar ahora tu mente podría ser “sin mirar la definición de `salario`, ¿cómo sé si es un atributo del objeto `persona`, o si por el contrario es un método que se aplica sobre el objeto `persona`?

La respuesta es: “no puedes saberlo”. En principio, no hay manera de determinar a priori si `salario` es un atributo, o un método que devuelve un valor. Pero también es cierto que no te hace falta saberlo: te basta con saber que `salario` es *algo* que devuelve el salario de una persona. Podría ser que `salario` fuese un atributo, y por tanto el salario sería algo almacenado como un dato interno del objeto. Pero también podría ser que `salario` fuese un método, y en ese caso, cada vez que solicitaras el salario de una persona, ese método se encargaría de recalcular de nuevo dicho salario a partir, posiblemente, de otros datos. En cualquier caso, desde el punto de vista de aquel que quiere saber el salario de la persona, le tiene sin cuidado el cómo se le proporciona la información solicitada. Le basta con saber que tiene que escribir `persona.salario`.

Esa ambigüedad atributo-método es intencionada, y se conoce como el **Principio del Acceso Uniforme**. La idea es que, si el salario se guarda ahora en un atributo, pero meses más tarde se necesita que sea calculado en lugar de almacenado, el cliente (es decir, el que utiliza el objeto `persona` con expresiones como `persona.salario`) no se entere de ese cambio, y no haya que modificar parte del programa por ese motivo.

A los atributos y métodos de un objeto, se les conoce en general como **características** del objeto, que en inglés se traduce como **features** (de ahí viene el que los atributos y los métodos de un objeto se declaren en la cláusula `feature` de su clase). Por tanto, un objeto estará compuesto por características, algunas de las cuales serán atributos y otras serán métodos.

## 8.5. Objetos como parámetros

Puedes pasar objetos como parámetros de la forma usual. Por ejemplo

```
imprimir_punto(p: PUNTO) is
  do
    print("(") print(p.x) print(", ") print(p.y) print(")")
  end
```

es un método que recibe un punto como argumento y lo imprime según el formato estándar. Si llamas a `imprimir_punto(blanco)`, imprimirá (3, 4).

Como un segundo ejemplo, podemos reescribir el método `distancia` del apartado 5.2 de forma que recibe dos PUNTOS como parámetros en lugar de cuatro DOUBLES.

```
distancia(p1, p2: PUNTO): DOUBLE is
  local
    dx, dy: DOUBLE
  do
    dx := p2.x - p1.x
```

```

dy := p2.y - p1.y
Result := (dx*dx + dy*dy).sqrt
end

```

## 8.6. Rectángulos

Los RECTANGULOS son parecidos a los puntos, excepto que tienen cuatro atributos, llamados *x*, *y*, *alto* and *ancho*.

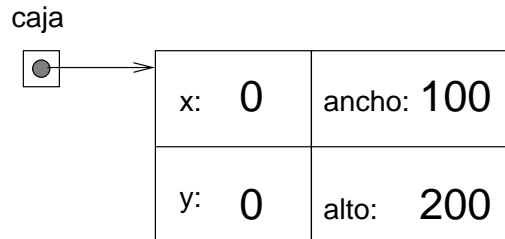
Por lo demás, todo es casi lo mismo.

```

local
  caja: RECTANGULO
do
  create caja.make(0, 0, 100, 200)
end

```

crea un nuevo objeto RECTANGULO y hace que *caja* haga referencia a él. La figura muestra el efecto de esta sentencia.



## 8.7. Objetos como tipos de retorno

Puedes escribir métodos que devuelvan objetos. Por ejemplo, *busca\_centro* recibe un RECTANGULO como argumento y devuelve un PUNTO que contiene las coordenadas del centro del rectángulo:

```

busca_centro(caja: RECTANGULO): DOUBLE is
  local
    x, y: DOUBLE
    p: PUNTO
  do
    x := caja.x + caja.ancho / 2
    y := caja.y + caja.alto / 2
    create p.make(x, y)
    Result := p
  end
end

```

Observa que hemos tenido que declarar una variable local de tipo PUNTO para poder crear el resultado y devolverlo. Para ahorrarnos esto, podemos usar una variante de la sentencia de creación de objetos, que en lugar de ser una sentencia es una **expresión de creación** de objetos, que devuelve directamente el objeto creado. El programa modificado quedaría:

```

busca_centro(caja: RECTANGULO): DOUBLE is
  local
    x, y: DOUBLE
  do
    x := caja.x + caja.ancho / 2
    y := caja.y + caja.alto / 2
    Result := create {PUNTO}.make(x, y)
  end

```

Observa que en este caso **create** no es una sentencia, sino una expresión, la cual puede usarse en la parte derecha de una asignación, pues devuelve el objeto creado en lugar de enlazar una referencia suya con una variable. Para usar esta variante, se debe incluir entre llaves el tipo del objeto que se quiere crear (en este caso, {PUNTO}).

## 8.8. Los objetos cambian

Puedes cambiar el contenido de un objeto desde fuera, pero para ello hay que dotar al objeto de los métodos adecuados que se encarguen de cambiar sus atributos por ti. Para ello, hay que definir un método dentro de la definición del tipo (o clase) del objeto. Por ejemplo, para poder cambiar el atributo **x** del rectángulo, definiremos un método (al que llamaremos **set\_x**) dentro de la clase **RECTANGULO**, que reciba un parámetro, y que su único cometido sea asignar el valor de ese parámetro al atributo **x**. Igualmente se hará con **y**. La clase final quedaría así:

```

class RECTANGULO

  creation make

  feature
    x, y, alto, ancho: DOUBLE

    set_x(xr: DOUBLE) is
      -- Asigna un valor a 'x'
    do
      x := xr
    end

    set_y(yr: DOUBLE) is
      -- Asigna un valor a 'y'
    do
      y := yr
    end

    make(xr, yr, alr, anr: DOUBLE) is
      -- Inicializar un objeto rectángulo
    do
      x := xr
      y := yr
      alto := alr
      ancho := anr
    end

end

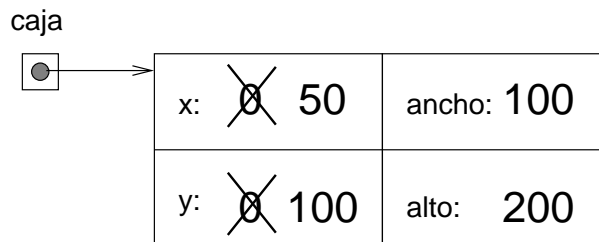
```

Así, por ejemplo, para “mover” un rectángulo sin cambiar su tamaño, puedes modificar los valores de `x` e `y` de la siguiente forma:

```
caja.set_x(caja.x + 50.0)
caja.set_y(caja.y + 100.0)
```

Como ves, la idea es solicitarle a la caja que sea ella la que cambie sus datos internos. Para ello, a través de las llamadas a los métodos `set_x` y `set_y` sobre `caja`, le solicitamos al objeto `caja` que cambie esos datos. Es decir: la caja *sabe* cómo cambiar sus datos, y así le pedimos que haga.

El resultado se muestra en la figura:



Podemos coger este código y encapsularlo en un método, y generalizarlo para que mueva el rectángulo todo lo que se quiera:

```
mover_rectangulo(caja: RECTANGULO; dx, dy: DOUBLE) is
do
    caja.set_x(caja.x + dx)
    caja.set_y(caja.y + dy)
end
```

Las variables `dx` y `dy` indican lo lejos que hay que mover el rectángulo en cada dirección. Invocar este método tiene el efecto de modificar el `RECTANGULO` que se pasa como argumento.

```
local
    caja: RECTANGULO
do
    create caja.make(0, 0, 100, 200)
    mover_rectangulo(caja, 50, 100)
    print("x = ") print(caja.x) print("%N")
    print("y = ") print(caja.y) print("%N")
    print("ancho = ") print(caja.ancho) print("%N")
    print("alto = ") print(caja.alto) print("%N")
end

imprime

x = 50.0
y = 100.0
ancho = 100.0
alto = 200.0
```

Modificar objetos pasándolos como argumentos a métodos puede ser útil, pero también hace más difícil la depuración porque no siempre está claro qué llamadas a métodos modifican o no modifican sus argumentos. Posteriormente discutiré los pros y contras de este estilo de programación.

En cualquier caso, parece más razonable hacer con `mover_rectangulo` lo mismo que hemos hecho con los métodos `set_x` y `set_y`, es decir, hacer que formen parte del tipo `RECTANGULO`. De esta forma, el rectángulo sabrá cómo moverse, y lo único que tendremos que hacer será pedirle que lo haga para nosotros. En ese caso, la clase `RECTANGULO` quedaría finalmente así:

```
class RECTANGULO

  creation make

  feature
    x, y, alto, ancho: DOUBLE

    set_x(xr: DOUBLE) is
      -- Asigna un valor a 'x'
      do
        x := xr
      end

    set_y(yr: DOUBLE) is
      -- Asigna un valor a 'y'
      do
        y := yr
      end

    mover(dx, dy: DOUBLE) is
      -- Mueve un rectángulo en ambas direcciones
      do
        x := x + dx
        y := y + dy
      end

    make(xr, yr, alr, anr: DOUBLE) is
      -- Inicializar un objeto rectángulo
      do
        x := xr
        y := yr
        alto := alr
        ancho := anr
      end
  end
end
```

Observa que, al incluir el método dentro de la definición de `RECTANGULO`, ya no es necesario incluir el objeto rectángulo como parámetro del método, pues éste quedará especificado cuando hagamos la invocación del método sobre dicho objeto, con la notación `caja.mover(50, 100)`. Igualmente, dentro de la definición del método `mover` se especifican los atributos `x` y `y` sin necesidad de indicar, con la notación punto, el objeto en cuestión (que en cada caso puede ser distinto). Es por ello por lo que la asignación `caja.x := caja.x + dx` se sustituye por `x := x + dx`.

De esta forma, el programa anterior quedaría:

```
local
  caja: RECTANGULO
do
  create caja.make(0, 0, 100, 200)
```

```

    caja.mover(caja, 50, 100)
    print("x = ") print(caja.x) print("%N")
    print("y = ") print(caja.y) print("%N")
    print("ancho = ") print(caja.ancho) print("%N")
    print("alto = ") print(caja.alto) print("%N")
end

```

Observa finalmente que hemos acertado el nombre `mover_rectangulo` para dejarlo simplemente en `mover`, pues ya no es necesario indicar qué se está moviendo (queda claro tan sólo mirando el tipo del objeto sobre el que se aplica el método: `RECTANGULO`).

## 8.9. Alias

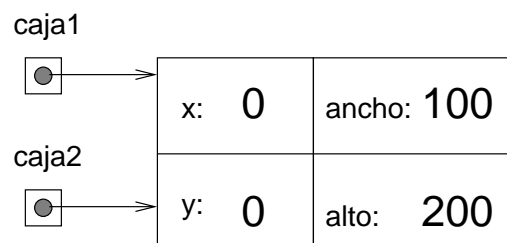
Recuerda que cuando haces una asignación a una variable objeto, estás asignando una *referencia* a un objeto. Es posible tener varias variables que se refieran al mismo objeto. Por ejemplo, este código:

```

local
    caja1, caja2: RECTANGULO
do
    create caja1.make(0, 0, 100, 200)
    caja2 := caja1
end

```

genera un diagrama de estado que se parece a éste:



Tanto `caja1` como `caja2` se refieren o “apuntan” al mismo objeto. En otras palabras, este objeto tiene dos nombres, `caja1` y `caja2`. Cuando una persona usa dos nombres, se dice que tiene un **alias**. Lo mismo ocurre con los objetos.

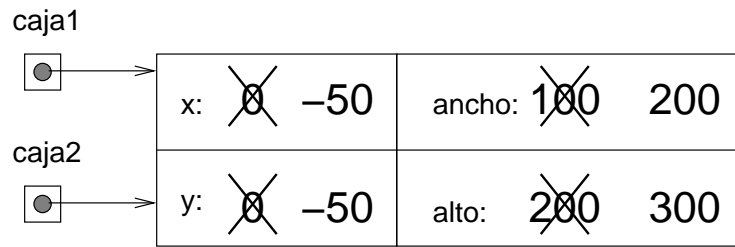
Cuando una variable es un alias para otra, todo cambio que afecte a una variable también afecta a la otra. Por ejemplo:

```

print(caja2.x)
caja1.mover(50, 50)
print(caja2.x)

```

La primera línea imprime 0, que es la posición `x` del rectángulo al que hace referencia `caja2`. La segunda línea cambia el valor de `x` (y de `y`) moviendo el rectángulo 50 puntos a la derecha y 50 puntos arriba. El efecto se muestra en la figura:



Como debe quedar claro al ver la figura, todos los cambios realizados a `caja1` también se aplican a `caja2`. Así, el valor impreso por la tercera línea es 50, la posición `x` del rectángulo movido.

Como puedes adivinar partiendo incluso de este sencillo ejemplo, el código que incluye alias puede volverse confuso rápidamente, y llegar a ser muy difícil de depurar. En general, se debe evitar el uso de los alias, o usarlos con precaución.

## 8.10. Void

Cuando creas una variable objeto, recuerda que estas creando una *referencia* a un objeto. Hasta que haces que la variable apunte a un objeto, el valor de esa variable es `Void`. `Void` es un valor especial de Eiffel (y una palabra reservada en Eiffel) que se utiliza para indicar que “no hay objeto”.

La declaración `blanco: PUNTO` es equivalente a esta inicialización:

```
local
  blanco: PUNTO
do
  blanco := Void
end
```

y se muestra en el siguiente diagrama de estado:

blanco



El valor `Void` se representa con un punto sin flecha.

Si intentas usar un objeto `Void`, accediendo a uno de sus atributos o invocando sobre él uno de sus métodos, obtendrás una excepción con el mensaje `"Target is Void"`, y el sistema finalizará el programa.

```
local
  blanco: PUNTO
  x: DOUBLE
do
  blanco := Void
  x := blanco.x          -- Target is Void.
  blanco.mover(50, 50)    -- Target is Void.
end
```

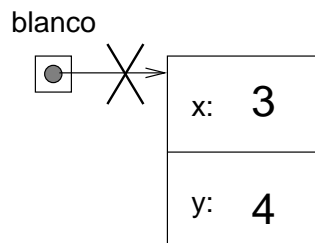
Por otra parte, está permitido pasar un objeto `Void` como argumento o recibir uno como valor de retorno. De hecho, es común hacerlo, por ejemplo, para representar un conjunto vacío o para indicar una condición de error.

## 8.11. Recogida de basura

En el apartado 8.9 hablamos sobre qué ocurre cuando más de una variable se refiere al mismo objeto. ¿Qué ocurre cuando *ninguna* variable se refiere a un objeto? Por ejemplo:

```
local
  blanco: PUNTO
do
  create blanco.make(3, 4)
  blanco := Void
end
```

La primera línea (tras `do`) crea un nuevo objeto `PUNTO` y hace que `blanco` se refiera a él. La segunda línea cambia `blanco` de manera que, en lugar de referirse al objeto, no se refiera a nada (el objeto `Void`).



Si no hay ninguna referencia a un objeto, entonces nadie puede leer o escribir ninguno de sus valores, o invocar alguno de sus métodos sobre él. En efecto, deja de existir. Podríamos mantener al objeto en memoria, pero lo único que haría sería gastar espacio, por lo que periódicamente, durante la ejecución del programa, el sistema Eiffel busca ese tipo de objetos “abandonados” y los reclama, en un proceso llamado **recogida de basura**. Después, el espacio de memoria ocupado por el objeto se hará disponible para ser usado como parte de un nuevo objeto.

No tienes que hacer nada para hacer que funcione la recogida de basura, y en general no debes preocuparte por ella.

## 8.12. Referencias y objetos expandidos

Existen dos tipos de objetos en Eiffel: unos se manipulan con referencias, y otros se manipulan directamente. A estos últimos se les denomina **objetos expandidos**.

Las diferencias entre unos y otros son:

**referencias:** Los objetos tales como `PUNTOS` y `RECTANGULOS` se manipulan a través de referencias a ellos. Eso provoca que si `x` e `y` son dos variables de tipo `PUNTO`, y hacemos que `y := x`, conseguiremos que ambas variables hagan referencia al mismo objeto (el fenómeno de los alias). También provoca el que un objeto pasado como argumento a un método se pueda cambiar desde dentro de ese método.

**objetos expandidos:** Los objetos tales como `INTEGERs` o `DOUBLEs` se manipulan directamente. Es decir: si `x` es una variable de tipo `DOUBLE` cuyo valor es 4.3, entonces en `x` estará almacenado el mismo valor 4.3, y NO una referencia al objeto 4.3. Por tanto, si tenemos que `y` es otra variable `DOUBLE`, y hacemos que `y := x`, al final `y` contendrá *otro* valor 4.3, distinto e independiente del valor 4.3 almacenado en `x`. Por eso, si cambiamos el valor de `x`, el valor de



y no quedará afectado en absoluto. Este es el comportamiento usual de las variables en los lenguajes de programación tradicionales.

Una muestra más de que el número 4.3 es un objeto en sí mismo, es que podemos solicitarle cosas directamente, como cuando le pedimos que se redondee al entero más próximo usando `(4.3).rounded`.

Por último, una diferencia más: una variable de un tipo expandido, si no se inicializa, asumirá automáticamente un valor por defecto que depende de su tipo: `0` para `INTEGERs`, `0.0` para `DOUBLEs`, `False` para `BOOLEANs`. Sin embargo, si la variable es de un tipo “referenciado”, su valor por defecto será `Void`, que representa la ausencia de objetos.

## 8.13. Glosario

**instancia:** Un ejemplo de una categoría. Mi gato es una instancia de la categoría “felinos”. Cada objeto es una instancia de alguna clase.

**atributo:** Uno de los ítems de datos con nombre que forman un objeto. Cada objeto (instancia) tiene su propia copia de los atributos de su clase.

**referencia:** Un valor que indica un objeto. En un diagrama de estado, una referencia se representa con una flecha.

**alias:** Cuando dos o más variables se refieren al mismo objeto.

**recogida de basura:** El proceso de buscar objetos que no tienen referencias y reclamar su espacio de almacenamiento.

**estado:** La descripción completa de todas las variables y objetos y sus valores, en un momento dado durante la ejecución de un programa.

**diagrama de estado:** Una instantánea del estado de un programa, mostrada gráficamente.

## Capítulo 9

# Crear tus propios objetos

### 9.1. Definiciones de clases y tipos de objetos

Cada vez que escribes una definición de clase, creas un nuevo tipo de objeto con el mismo nombre que la clase. En el capítulo anterior vimos dos claros ejemplos de ello: las clases `PUNTO` y `RECTANGULO`. Una vez definidas, disponemos ya de los tipos `PUNTO` y `RECTANGULO` como si de otros tipos predefinidos se tratasen (`INTEGER`, `DOUBLE`, etcétera). Gracias a ello, pudimos declarar variables de tipo `PUNTO`, y usar puntos de forma análoga a como usamos enteros, por ejemplo.

He aquí las ideas más importantes de este capítulo:

- Definir una nueva clase también supone crear un nuevo tipo de objeto con el mismo nombre. De hecho, en Eiffel no hay “tipos” como tales, sino que todo son clases.
- Una definición de clase es como una plantilla para crear objetos: determina qué atributos tienen los objetos y qué métodos pueden actuar sobre ellos.
- Cada objeto pertenece a un cierto tipo; por tanto, es una instancia de una cierta clase.
- Cuando creas un objeto con la sentencia `create`, al mismo tiempo invocas a un método especial al que hasta ahora hemos estado llamando `make`. Ese método se denomina **método de creación** o **constructor**. Como dijimos en el capítulo anterior, podemos usar el método `make` en una sentencia `create` porque el nombre `make` aparece dentro de la cláusula `creation` de la clase. Eso significa que una clase puede tener varios métodos de creación simplemente enumerándolos todos en la cláusula `creation`. También significa que el método de creación no tiene necesariamente que llamarse `make`.
- Lo normal es que todos los métodos que trabajan con un tipo van en la definición de la clase correspondiente. Eso es algo que ya hicimos en el capítulo anterior con el método `mover_rectangulo`, el cual colocamos dentro de la clase `RECTANGULO`, pasando a llamarse `mover`.

He aquí algunas cuestiones sintácticas sobre las definiciones de clases:

- Los nombres de las clases (y por tanto, de los tipos de objetos) siempre se escriben en mayúsculas, como `INTEGER` o `PUNTO`.
- Se debe cada definición de clase en un fichero separado, y el nombre del fichero debe coincidir con el nombre de la clase, en minúsculas, con el sufijo `.e`. Por ejemplo, la clase `HOLA` debe ir definida en el fichero `hola.e`.

- En todo programa, una clase queda destacada como la **clase raíz**. Dentro de ésta, se destaca uno de sus métodos como el **método raíz**. (Por ejemplo, en el programa “¡Hola, mundo!” la clase raíz era **HOLA** y el método raíz era **make**.) El método raíz es el lugar donde comienza la ejecución del programa. No existe ninguna palabra clave especial para decir cuál es la clase raíz y cuál es el método raíz, pero el compilador debe saber cuáles son para poder construir correctamente el programa. Para ello hay varias técnicas, pero la más sencilla consiste en escribir los nombres de la clase y el método raíz en un formulario del entorno de desarrollo utilizado para editar y compilar los programas. Eso es algo que depende del entorno que utilices, así que lo mejor es que consultes la documentación del mismo.

Normalmente no tiene mucho sentido crear objetos que sean instancias de la clase raíz, y no está muy claro por qué sería bueno hacerlo.

Con todos estos asuntos aclarados, echemos un vistazo al ejemplo de otro tipo definido por el usuario: **HORA**.

## 9.2. Hora

Una motivación frecuente para crear un nuevo tipo de objeto es tomar varias porciones de datos relacionados y encapsularlos en un objeto que se pueda manipular (pasar como argumento, operar sobre él) como una unidad. Ya hemos visto dos tipos así: **PUNTO** y **RECTANGULO**.

Otro ejemplo que nosotros mismos vamos a implementar es **HORA**, que usaremos para registrar la hora del día. Los trozos de información que forman una hora son la hora, los minutos y los segundos. Como cada objeto **HORA** contiene esos datos, necesitamos crear atributos para almacenarlos.

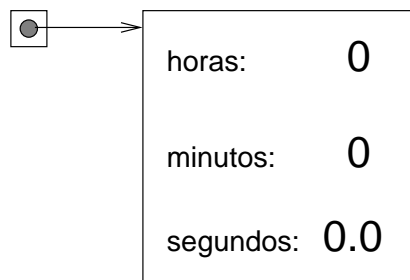
El primer paso es decidir qué tipo debería tener cada atributo. Parece claro que **horas** y **minutos** deberían ser enteros. Sólo para hacer las cosas interesantes, consideremos que **segundos** es un **DOUBLE**, por lo que podremos registrar fracciones de segundo.

Los atributos de una clase, al igual que los métodos, son características de esa clase, y como tales, se declaran en la cláusula **feature** de la clase, así:

```
class HOLA

feature
  horas, minutos: INTEGER
  segundos: DOUBLE
end
```

En sí, este fragmento de código es una correcta definición de clase. El diagrama de estado para un objeto **HORA** podría ser así:



Después de declarar los atributos, el siguiente paso normalmente es definir un constructor para la nueva clase.

### 9.3. Constructores

La utilidad normal de un constructor es inicializar los atributos. Un constructor es un método igual que cualquier otro, con estas dos diferencias:

- Su nombre debe aparecer en la cláusula **creation** de la clase.
- No deben devolver ningún valor de retorno, y por tanto, no deben declararse con un tipo de retorno.

Si bien el constructor puede llamarse como quieras, lo normal y más recomendable es que sigas el convenio de llamarlo **make**.

He aquí un ejemplo para la clase **HORA**:

```
class HORA

  creation make

  feature
    horas, minutos: INTEGER
    segundos: DOUBLE

  make is
    -- Inicializa un objeto hora.
  do
    horas := 0
    minutos := 0
    segundos := 0.0
  end
end
```

Este constructor no recibe argumentos. Cada línea del constructor inicializa un atributo a un valor por defecto arbitrario (en este caso, medianoche).

### 9.4. Más constructores

Como comentamos antes, una clase puede tener varios constructores, simplemente enumerándolos en la cláusula **creation**.

Es muy común tener un constructor que no recibe parámetros (como el visto arriba), y otro constructor que recibe una lista de parámetros idéntico a la lista de atributos. Por ejemplo:

```
class HORA

  creation
    make,
    make_par

  feature
    horas, minutos: INTEGER
    segundos: DOUBLE
```

```

make is
    -- Inicializa un objeto hora.
do
    horas := 0
    minutos := 0
    segundos := 0.0
end

make_par(h, m: INTEGER; s: DOUBLE) is
    -- Inicializa la hora con parámetros.
do
    horas := h
    minutos := m
    segundos := s
end
end

```

Observa que ahora la cláusula **creation** contiene dos nombres, que son **make** y **make\_par**. Estos dos métodos serán los constructores de la clase **HOLA**.

Los tipos de los parámetros son los mismos que los de los correspondientes atributos. Todo lo que hace el constructor es copiar la información de los parámetros a los atributos.

## 9.5. La variable especial **Current**

**Current** es una variable especial (al estilo de **Result**) que siempre está disponible en cualquier método, y que representa al objeto actual, es decir, el objeto sobre el que se aplica el método en un momento determinado.

Usando **Current** podemos referirnos a un atributo usando la misma notación punto que utilizamos cuando queremos referirnos a un atributo de otro objeto:

```

class HORA
    -- ...
feature
    -- ...
    imprimir_la_hora is
        -- Muestra la hora por pantalla.
    do
        print(Current.horas)
        print(":")
        print(Current.minutos)
        print(":")
        print(Current.segundos)
        print(":")
    end
end
end

```

Aquí también podríamos haber usado simplemente **horas** en lugar de **Current.horas**, porque en ambos casos estamos accediendo al atributo del objeto actual. ¿Para qué, entonces, vamos a usar **Current** si tenemos que escribir más? Sobre todo, por legibilidad: se deja más claro el hecho de que **horas** es un atributo del objeto actual.

Otra utilidad sería la de comprobar si el objeto que se pasa como parámetro a un método es precisamente el objeto actual:

```

class HORA
  -- ...
feature
  -- ...
  comprobar_hora(h: HORA) is
    do
      if Current = h then
        print("Son la misma hora")
      else
        print("Son horas diferentes")
      end
    end
  end
end

```

Si probamos el método de esta forma:

```

class RAIZ
  creation make

feature
  make is
    local
      h1, h2: HORA
    do
      create h1.make
      create h2.make
      h1.comprobar_hora(h2)
      h1.comprobar_hora(h1)
    end
  end
end

```

El resultado sería

```

Son horas diferentes
Son la misma hora

```

`Current` es más útil de lo que parece a simple vista, pero eso es algo que demostraremos más tarde.

## 9.6. Crear un nuevo objeto

Aunque los constructores son métodos, normalmente no los invocarás directamente. En su lugar, se suelen usar dentro de sentencias `create`. De esta forma, primero se reserva espacio para el nuevo objeto y luego se invoca al constructor para inicializar los atributos.

El siguiente programa ilustra dos maneras de crear e inicializar objetos `HORA`:

```

class HORA

  creation
    make,

```

```

        make_par

feature
    horas, minutos: INTEGER
    segundos: DOUBLE

    make is
        -- Inicializa un objeto hora.
        do
            horas := 0
            minutos := 0
            segundos := 0.0
        end

    make_par(h, m: INTEGER; s: DOUBLE) is
        -- Inicializa la hora con parámetros.
        do
            horas := h
            minutos := m
            segundos := s
        end
    end

end

class PRINCIPAL

creation make

feature
    make is
        -- Programa principal.
        local
            h1, h2: HORA
        do
            -- una manera de crear e inicializar un objeto HORA
            create h1.make
            print(h1) print("%N")
            -- otra forma de hacerlo
            create h2.make_par(11, 8, 3.14159)
            print(h2)
        end
    end

end

```

La clase raíz es `PRINCIPAL`, y el método raíz es el método `make` de `PRINCIPAL`. Recuerda que cada clase debe ir en un fichero distinto. Como ejercicio, sigue el flujo de ejecución de este programa.

En el método raíz, la primera vez que construimos un objeto `HORA` usamos `make`, por lo que no pasamos parámetros. En la siguiente línea construimos otro objeto `HORA`, pero esta vez usando `make_par`, y en este caso le pasamos los parámetros necesarios.

## 9.7. Imprimir un objeto

La salida de este programa es:

```
HORA#0x8062c10 []
HORA#0x8062c28 []
```

Cuando Eiffel imprime el valor de un objeto cuyo tipo está definido por el usuario, imprime el nombre del tipo y un código hexadecimal (base 16) especial que es único para cada objeto. Este código no tiene ningún significado en sí mismo; de hecho, puede variar de máquina en máquina e incluso de ejecución en ejecución. Pero puede ser útil durante la depuración, en el caso de que quieras seguir la pista a los objetos individuales.

Con idea de imprimir los objetos de forma que tengan más significado para los usuarios (en lugar de para los programadores), normalmente querrás escribir un método llamado algo así como `imprimir_hora`:

```
imprimir_hora(h: HORA) is
do
    print(h.horas) print(":")
    print(h.minutos) print(":")
    print(h.segundos) print("%N")
end
```

La salida de este método, si pasas `h2` como argumento, es `11:8:3.14159`. Aunque esto puede interpretarse como una hora, no está realmente en un formato estándar. Por ejemplo, si el número de minutos o segundos es menor que 10, esperaríamos un 0 a la izquierda de la cantidad. Además, podemos querer cortar la parte decimal de los segundos. En otras palabras, queremos algo como `11:08:03`.

En muchos lenguajes, existen formas simples de controlar el formato de salida de los números. En Eiffel no hay formas simples, en general, pero si estás interesado en mostrar fechas y horas en un formato inteligible, te recomiendo que eches un vistazo a las clases `TIME` y `TIME_IN_SPANISH` de la librería base de Eiffel.

## 9.8. Operaciones sobre objetos

Aunque no podemos imprimir horas en un formato óptimo, todavía podemos escribir métodos que manipulen objetos `HORA`. En los próximos apartados, mostraré varios de los posibles interfaces de métodos que operan con objetos. Para algunas operaciones, podrás elegir varias interfaces posibles, por lo que deberías considerar los pros y contras de cada uno de ellos:

**función pura:** Recibe objetos como argumentos pero no los modifica. El valor de retorno será un nuevo objeto creado dentro del método.

**modificador:** Recibe objetos como argumentos y modifica algunos de ellos. A menudo no devuelven nada.

**método rellenedor:** Uno de los argumentos es un objeto “vacío” que se rellena dentro del método. Técnicamente, es un tipo de modificador.

## 9.9. Funciones puras

Un método se considera función pura si el resultado depende sólo de sus argumentos, y no tienen efectos laterales como modificar un argumento o imprimir algo. El único resultado de invocar una función pura es el valor de retorno.

Un ejemplo es `despues`, que compara dos `HORAS` y devuelve un valor lógico que indica si el primer operando viene antes que el segundo:



```

despues(h1, h2: HORA): BOOLEAN is
do
  if h1.horas > h2.horas then
    Result := True
  elseif h1.horas < h2.horas then
    Result := False
  else
    if h1.minutos > h2.minutos then
      Result := True
    elseif h1.minutos < h2.minutos then
      Result := False
    else
      if h1.segundos > h2.segundos then
        Result := True
      else
        Result := False
      end
    end
  end
end
end

```

¿Cuál es el resultado de este método si las dos horas son iguales? ¿Parece el resultado apropiado para este método? Si estuvieras escribiendo la documentación de este método, ¿mencionarías ese caso específicamente?

Un segundo ejemplo es `suma_horas`, que calcula la suma de dos horas. Por ejemplo, si son las 9:14:30, y tu tostadora tarda 3 horas y 35 minutos, puedes usar `suma_horas` para hacerte una idea de cuándo estarán listas las tostadas.

Aquí tenemos un esbozo de este método, que no está del todo correcto:

```

suma_horas(h1, h2: HORA): HORA is
local
  suma: HORA
do
  create suma.make
  suma.set_horas(h1.horas + h2.horas)
  suma.set_minutos(h1.minutos + h2.minutos)
  suma.set_segundos(h1.segundos + h2.segundos)
  Result := suma
end

```

Observa los métodos `set_horas`, `set_minutos` y `set_segundos`. Son análogos al método `set_x` de la clase `RECTANGULO`: lo que hacen es asignar al atributo correspondiente el valor pasado como argumento. Como ejercicio, defínelos.

Aunque este método devuelve un objeto `HORA`, no es un constructor. Tan sólo hay que ver la diferencia sintáctica que tiene con los constructores.

Aquí tenemos un ejemplo de cómo usar este método. Si `hora_actual` contiene la hora actual y `hora_tostada` contiene la cantidad de tiempo que le lleva a tu tostadora preparar tostadas, entonces podrías usar `sumar_horas` para saber cuándo estará lista la tostada.

```

local
  hora_actual, hora_tostada, hora_final: HORA
do

```

```

hora_actual := create {HORA}.make_par(9, 14, 30.0)
hora_tostada := create {HORA}.make_par(3, 35, 0.0)
hora_final := sumar_horas(hora_actual, hora_tostada)
imprimir_hora(hora_final)
end

```

La salida de este programa es 12:49:30.0, lo cual es correcto. Por otra parte, hay casos en los cuales el resultado es correcto. ¿Sabrías indicar uno?

El problema es que este método no es capaz de tratar con los casos donde el número de segundos o minutos suman más de 60. En este, debemos “acarrear” los segundos extra en la columna de los minutos, los minutos extra en la columna de las horas.

Esta es una segunda versión, ya corregida, de este método.

```

sumar_horas(h1, h2: HORA): HORA is
  local
    suma: HORA
  do
    create suma.make
    suma.set_horas(h1.horas + h2.horas)
    suma.set_minutos(h1.minutos + h2.minutos)
    suma.set_segundos(h1.segundos + h2.segundos)

    if suma.segundos >= 60.0 then
      suma.set_segundos(suma.segundos - 60.0)
      suma.set_minutos(suma.minutos + 1)
    end

    if suma.minutos >= 60 then
      suma.set_minutos(suma.minutos - 60)
      suma.set_horas(suma.horas + 1)
    end

    Result := suma
  end
end

```

Aunque es correcto, la cosa comienza a hacerse grande. Más tarde sugeriré una aproximación alternativa a este problema que será mucho más corta.

## 9.10. Modificadores

Como un ejemplo de modificador, considera **incrementar**, que suma cierto número de segundos a un objeto **HORA**. De nuevo, un esbozo de este método podría ser así:

```

incrementar(h: HORA; segs: DOUBLE) is
  do
    h.set_segundos(h.segundos + segs)
    if h.segundos >= 60.0 then
      h.set_segundos(h.segundos - 60.0)
      h.set_minutos(h.minutos + 1)
    end
    if h.minutos >= 60 then

```

```

        h.set_minutos(h.minutos - 60)
        h.set_horas(h.horas + 1)
    end
end

```

La primera línea realiza la operación básica; las restantes tratan los mismos casos que vimos antes.

¿Es correcto este método? ¿Qué ocurre si el argumento **segs** es mucho mayor que 60? En ese caso, no es suficiente con restar 60 una vez; tenemos que seguir haciéndolo hasta que **segundos** quede por debajo de 60. Podemos hacerlo simplemente sustituyendo las sentencias **if** por bucles **loop**:

```

incrementar(h: HORA; segs: DOUBLE) is
do
    h.set_segundos(h.segundos + segs)

    from
    until
        h.segundos < 60
    loop
        h.set_segundos(h.segundos - 60.0)
        h.set_minutos(h.minutos + 1)
    end

    from
    until
        h.minutos < 60
    loop
        h.set_minutos(h.minutos - 60)
        h.set_horas(h.horas + 1)
    end
end
end

```

Esta solución es correcta, pero no muy eficiente. ¿Puedes pensar en una solución que no requiera iteración?

## 9.11. Métodos rellenos

En ocasiones veremos métodos como **sumar\_horas** escritos con diferentes interfaces (diferentes argumentos y valores de retorno). En lugar de crear un nuevo objeto cada vez que se invoca a **sumar\_horas**, podemos pedirle al llamante que proporcione un objeto “vacío” donde **sumar\_horas** pueda almacenar el resultado. Compara lo siguiente con la versión anterior:

```

sumar_horas_relleno(h1, h2, suma: HORA) is
do
    suma.set_horas(h1.horas + h2.horas)
    suma.set_minutos(h1.minutos + h2.minutos)
    suma.set_segundos(h1.segundos + h2.segundos)
    if suma.segundos >= 60.0 then
        suma.set_segundos(suma.segundos - 60.0)
        suma.set_minutos(suma.minutos + 1)
    end
    if suma.minutos >= 60 then

```

```

        suma.set_minutos(suma.minutos - 60)
        suma.set_horas(suma.horas + 1)
    end
end

```

Una ventaja de este enfoque es que el llamante tiene la opción de reutilizar el mismo objeto repetidamente para realizar una serie de sumas. Esto puede resultar ligeramente más eficiente, aunque también lo bastante confuso como para causar sutiles errores. Para la gran mayoría de las tareas de programación, merece la pena perder un poco de tiempo de ejecución con idea de evitar un montón de tiempo de depuración.

## 9.12. ¿Cuál es mejor?

Todo lo que puede hacerse con modificadores y funciones rellenas también puede hacerse con funciones puras. De hecho, existen lenguajes de programación, llamados lenguajes **funcionales**, que sólo admiten funciones puras. Algunos programadores creen que los programas que usan funciones puras son más rápidos de desarrollar y menos propensos a errores que los programas que usan modificadores. No obstante, hay veces en los que los modificadores son más convenientes, y veces en los que los programas funcionales son menos eficientes.

En general, te recomiendo que escribas funciones puras mientras sea razonable hacerlo, y adoptar el recurso de los modificadores únicamente si conllevan una clara ventaja. Este enfoque podría llamarse un estilo de programación funcional.

## 9.13. Desarrollo incremental contra planificación

En este capítulo he mostrado un enfoque para el desarrollo de programas al que llamaré **prototipado rápido con mejora iterativa**. En cada caso escribí un esbozo (o prototipo) que realizaba el cálculo básico, y luego lo probé para unos cuantos casos, corrigiendo los fallos a medida que los iba encontrando.

Aunque este enfoque puede ser efectivo, puede llevar a obtener código innecesariamente complicado —puesto que trata muchos casos especiales— e inseguro —pues te resulta difícil convencerte a ti mismo que has encontrado *todos* los errores.

Una alternativa es la planificación de alto nivel, en la cual un poco de perspicacia sobre el problema puede hacer la programación mucho más sencilla. ¡En este caso la perspicacia es que una **HORA** es en realidad un número de tres dígitos en base 60! **segundos** es la “columna de las unidades”, **minutos** es la “columna de los 60s” y **horas** es la “columna de los 3600s”.

Cuando escribimos **sumar\_horas** e **incrementar**, efectivamente realizamos sumas en base 60, que es por lo cual tenemos que arrastrar un “acarreo” de una columna a la siguiente.

Por lo tanto, un enfoque alternativo al problema global es convertir **HORAS** en **DOUBLES** y aprovechar la ventaja de que el ordenador ya sabe realizar operaciones aritméticas con **DOUBLES**. Este es un método que convierte una **HORA** en un **DOUBLE**:

```

convertir_a_segundos(h: HORA): DOUBLE is
    local
        minutos: INTEGER
        segundos: DOUBLE
    do
        minutos := h.horas * 60 + h.minutos
        segundos := minutos * 60 + h.segundos
    end
end

```

```

    Result := segundos
end

```

Ahora todo lo que necesitamos es una manera de convertir un `DOUBLE` en un objeto `HORA`. Podemos escribir un método para ello, y lo más adecuado es que éste se convierta en el tercer constructor, para lo cual es necesario que su nombre aparezca en la cláusula `creation` de `HORA`:

```

make_segs(segs: DOUBLE) is
  local
    s: DOUBLE
  do
    s := segs
    horas := s // 3600.0
    s := s - horas * 3600.0
    minutos := s // 60.0
    s := s - minutos * 60
    segundos := s
  end
end

```

Este constructor es un poco diferente de los otros, ya que implica realizar algunos cálculos junto con asignaciones a los atributos.

Debes pensar un poco para convencerte de que esta técnica que estoy usando para convertir una base en otra es correcta. Suponiendo que estés convencido, podemos usar estos métodos para reescribir `sumar_horas`:

```

sumar_horas(h1, h2: HORA): HORA is
  local
    segundos: DOUBLE
  do
    segundos := convertir_a_segundos(h1) + convertir_a_segundos(h2)
    Result := create {HORA}.make_segs(segundos)
  end
end

```

Esto queda mucho más corto que la versión original, y además es más fácil demostrar que es correcto (suponiendo, como es usual, que los métodos se invocan de forma correcta). Como ejercicio, reescribe `incrementar` de la misma manera.

## 9.14. Generalización

En cierta forma convertir de la base 60 a la base 10 y al revés es más difícil que trabajar con simples horas. La conversión de bases es más abstracta; nuestra intuición para trabajar con horas es mejor.

Pero si tenemos la perspicacia de trabajar con horas como si fuesen números en base 60, y realizamos la inversión de escribir los métodos de conversión (`convertir_a_segundos` y el tercer constructor), tendremos un programa que es más corto, más fácil de leer y depurar, y más fiable.

También es más sencillo añadir características nuevas posteriormente. Por ejemplo, imagina restar dos `HORAS` para saber la duración entre ellas. El enfoque “ingenuo” podría ser implementar la resta mediante un “acarreo negativo”. Usando los métodos de conversión podría ser mucho más fácil.

Irónicamente, a veces hacer un problema más difícil (más general) lo convierte en más fácil (menos casos especiales, menos oportunidades de error).

## 9.15. Algoritmos

Cuando escribes una solución general para toda una clase de problemas, en lugar de una solución específica para un sólo problema, has escrito un **algoritmo**. Ya mencioné esta palabra en el capítulo 1, pero no la definí de forma precisa. No es fácil de definir, así que intentaré un par de aproximaciones.

Primero, considera algunas cosas que no son algoritmos. Por ejemplo, cuando aprendiste a multiplicar números de un sólo dígito, probablemente memorizaste la tabla de multiplicar. En efecto, memorizas 100 soluciones específicas, por lo que el conocimiento no es realmente algorítmico.

Pero si fueras “perezoso”, probablemente harías trampa aprendiendo unos cuantos trucos. Por ejemplo, para calcular el producto de  $n$  por 9, puedes escribir  $n - 1$  como el primer dígito y  $10 - n$  como el segundo dígito. Este truco es una solución general para multiplicar cualquier número de un sólo dígito, por 9. ¡Eso sí es un algoritmo!

Igualmente, todas las técnicas que aprendiste para sumar con acarreo, restar con acarreo negativo, y dividir números grandes, son algoritmos. Una de las características de los algoritmos es que no requieren ninguna inteligencia para seguirlos. Son procesos mecánicos en los que cada paso viene del anterior de acuerdo a un sencillo conjunto de reglas.

En mi opinión, es embarazoso el que los humanos gasten mucho tiempo en la escuela aprendiendo a ejecutar algoritmos que, casi literalmente, no requieren inteligencia.

Por otra parte, el proceso de diseñar algoritmos es interesante, intelectualmente desafiante, y una parte central de lo que llamamos programación.

Algunas de las cosas que la gente hace de forma natural, sin dificultad ni pensamiento consciente, son las más difíciles de expresar algorítmicamente. Entender el lenguaje natural es un buen ejemplo. Todos lo hacemos, pero hasta ahora nadie ha sido capaz de explicar *cómo* lo hacemos, al menos no en forma de algoritmo.

Más tarde tendrás oportunidad de diseñar algoritmos sencillos para una variedad de problemas.

## 9.16. Glosario

**clase:** Previamente, definí una clase como una colección de métodos relacionados. En este capítulo aprendimos que una clase es también una plantilla para un nuevo tipo de objeto.

**instancia:** Un miembro de una clase. Cada objeto es una instancia de una cierta clase.

**constructor:** Un método especial que inicializa los atributos de un objeto recién construido.

**sistema:** Una colección de una o más definiciones de clases (una por fichero) que forman un programa.

**clase raíz:** La clase que contiene el método raíz.

**método raíz:** El método, definido en la clase raíz, por el que comienza la ejecución del programa.

**función:** Un método cuyo resultado depende únicamente de sus parámetros, y que no tiene efectos laterales más que el de devolver un valor.

**estilo de programación funcional:** Un estilo de diseño de programas en el que la mayoría de los métodos son funciones

**modificador:** Un método que cambia uno o más objetos que recibe como parámetro, y que normalmente no devuelve nada.

**método rellenador:** Un tipo de método que recibe un objeto “vacío” como parámetro y rellena sus atributos, en lugar de generar un valor de retorno. Este tipo de método no es normalmente la mejor opción.

**algoritmo:** Un conjunto de instrucciones para resolver una clase de problemas por medio de un proceso mecánico y no inteligente.

## Capítulo 10

# Arrays

Un **array** es un conjunto de valores en el cual cada valor queda identificado por un índice. Puedes tener un array de **INTEGERs**, **DOUBLEs**, o cualquier otro tipo, pero todos los valores en un array tienen que tener el mismo tipo.

Puedes declarar variables de tipo array de la siguiente forma:

```
local
  cuenta: ARRAY[INTEGER]
  valores: ARRAY[DOUBLE]
do
end
```

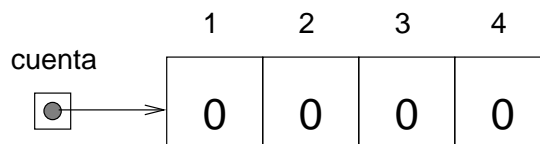
Observa la sintaxis. Para declarar que **cuenta** es un array de enteros, escribimos la palabra **ARRAY**, y a continuación, entre corchetes, la palabra **INTEGER**. Así se declara siempre un array: indicando entre corchetes el tipo de sus elementos. Por la misma razón, **valores** es un array cuyos elementos son de tipo **DOUBLE**.

Hasta que las inicializas, esas variables quedan asignadas a **Void**. Para crear el array en sí mismo, usa la sentencia **create** combinada con la llamada al constructor **make**.

```
local
  cuenta: ARRAY[INTEGER]
  valores: ARRAY[DOUBLE]
do
  create cuenta.make(1, 4)
  create valores.make(1, tamaño)
end
```

La primera sentencia hace que **cuenta** haga referencia a un array de 4 enteros, donde el primer entero tiene el índice 1, y el último tiene el índice 4; la segunda sentencia hace que **valores** haga referencia a un array de **DOUBLEs**. El número de elementos que almacena **valores** depende de **tamaño**. Puedes usar cualquier expresión entera a la hora de indicar cuál será el índice menor o el mayor del array que estás creando. Incluso pueden ser negativos.

La figura siguiente muestra cómo se representan los arrays en un diagrama de estado:





Los números grandes dentro de las cajas son los **elementos** del array. Los números pequeños fuera de las cajas son los índices usados para identificar a cada caja. Cuando se crea un nuevo array, los elementos se inicializan al valor por defecto del tipo de los elementos. En este caso, el valor por defecto del tipo `INTEGER` es cero.

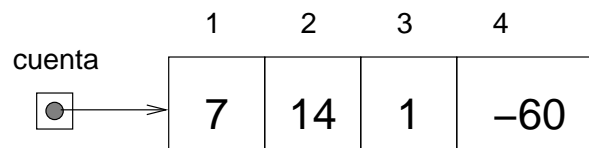
## 10.1. Acceder a los elementos

Cuando se accede a un elemento del array puede ser por dos motivos: porque se quiere almacenar un valor en dicha posición, o porque se desea saber qué valor se encuentra almacenado en dicha posición. Eiffel dispone de un método distinto para cada caso.

Si quieres saber qué valor tiene un elemento del vector, utiliza el operador `@`. Por ejemplo, `cuenta @ 1` devuelve el valor del primer elemento del array, y `cuenta @ 2` devuelve el segundo. Para almacenar valores en un array, utiliza el método `put`. Por ejemplo:

```
cuenta.put(7, 1)
cuenta.put((cuenta @ 1) * 2, 2)
cuenta.put((cuenta @ 3) + 1, 3)
cuenta.put((cuenta @ 4) - 60, 4)
```

Este es el efecto del fragmento de código anterior:



Como puedes observar, `cuenta.put(7, 1)` almacena el número 7 en el elemento n° 1 del array `cuenta`. Y `cuenta.put((cuenta @ 1) * 2, 2)` almacena en el segundo elemento del array `cuenta`, el resultado de multiplicar por dos el valor almacenado en el primer elemento del array.

Aquí, los cuatro elementos del array se identifican con los índices 1, 2, 3 y 4. Esto es así porque en la creación del array se dijo que el índice mínimo del array es 1, y que el índice máximo es 4. Sin embargo, también podría haberse creado el array de tal manera que el primer índice fuese -3, y que el máximo fuese 0, quedando los índices -3, -2, -1 y 0. En ambos casos, tenemos un array con cuatro elementos.

En todo caso, si se intenta acceder a un elemento que no existe (es decir, un elemento cuyo índice se encuentre fuera del intervalo entre el índice mínimo y el máximo), se producirá una excepción, mostrando un mensaje de error y finalizando el programa.

Puedes usar cualquier expresión como índice, siempre que su valor sea de tipo `INTEGER`. Una de las maneras más típicas de indexar un array es usar un contador en un bucle. Por ejemplo:

```
local
  i: INTEGER
  cuenta: ARRAY[INTEGER]
do
  create cuenta.make(1, 4)

  from
    i := 1
  until
    i > 4
```

```

    loop
        print(cuenta @ i) print("%N")
        i := i + 1
    end
end

```

Este es un bucle estándar que cuenta desde 1 hasta 4, y cuando la variable `i` es 5, la condición se cumple y el bucle finaliza. De esta forma, el cuerpo del bucle se ejecuta sólo cuando `i` es 1, 2, 3 y 4.

En cada iteración del bucle usamos `i` como índice para el array, imprimiendo el valor del elemento `i`-ésimo. Este tipo de recorrido de arrays es muy común. Los arrays y los bucles van juntos como los langostinos y la manzanilla.

## 10.2. lower, upper y count

Los arrays disponen de métodos y atributos propios que proporcionan información sobre ellos mismos. El método `lower` aplicado a un array devuelve su índice mínimo (es decir, el índice más pequeño donde hay almacenado un elemento). El método `upper` devuelve su índice máximo. Y el método `count` devuelve el número de elementos que tiene el array.

Con los métodos `lower` y `upper` podemos reescribir el bucle anterior de manera que tengamos que recordar cuáles eran los índices mínimo y máximo del array:

```

local
    i: INTEGER
    cuenta: ARRAY[INTEGER]
do
    create cuenta.make(1, 4)

    from
        i := cuenta.lower
    until
        i > cuenta.upper
    loop
        print(cuenta @ i) print("%N")
        i := i + 1
    end
end
end

```

Ahora el bucle, en lugar de usar 1 y 4, utiliza `cuenta.lower` y `cuenta.upper`, respectivamente. Así nos queda un esquema general de recorrido de arrays, independientemente de los índices exactos que se utilicen en cada array en particular.

En este caso, `cuenta.count` vale 4, porque el array tiene cuatro elementos. Aquí ha dado la casualidad de que su valor coincide con `cuenta.upper`, pero eso no siempre tiene que ser siempre así. Si el array se hubiera creado con la sentencia `create.make(-2, 1)`, tendríamos un array donde `cuenta.lower` vale -2, `cuenta.upper` vale 1 y `cuenta.count` vale 4.

De hecho, siempre se cumple que  $count = upper - lower + 1$ .

## 10.3. Copiar arrays

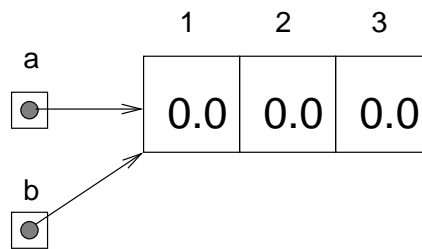
Cuando copias una variable array, recuerda que estás copiando una referencia al array. Por ejemplo:

```

local
  a, b: ARRAY[DOUBLE]
do
  create a.make(1, 3)
  b := a
end

```

Este código crea un array de tres `DOUBLE`s, y hace que dos variables diferentes hagan referencia a él. Esta situación es como la que se da con los alias.



Todo cambio que se haga en uno de los arrays se reflejará en el otro. Este no es normalmente el comportamiento que quieres; en su lugar, debes hacer una copia del array, creando un nuevo array y copiando cada elemento de uno al otro.

```

local
  a, b: ARRAY[DOUBLE]
  i: INTEGER
do
  create b.make(1, 3)
  from
    i := 0
  until
    i > 4
  do
    b.put(a @ i, i)
    i := i + 1
  end
end
end

```

Una solución sin tener que escribir un bucle es la de usar el método `copy`, el cual es válido siempre que los elementos del array pertenezcan a tipos **expandidos**, como `INTEGER` o `DOUBLE` (pero no como `STRING`, pues sus objetos son *referencias*). Si los elementos del array fuesen referencias, en lugar de `copy` se deberá usar `deep_copy`. (Recuerda lo que hablamos sobre referencias y objetos expandidos en el apartado 8.12.)

Como en este ejemplo se usa un array de `DOUBLE`s, y como los objetos `DOUBLE` son objetos expandidos, usaremos el método `copy`, y no `deep_copy`:

```

local
  a, b: ARRAY[DOUBLE]
do
  create b.make(1, 3)
  b.copy(a)
end

```

Este trozo de código es equivalente al anterior: lo que hace es copiar en el array **b** todos los elementos del array **a**, por lo que tras la ejecución del método **copy**, los arrays **a** y **b** serán arrays distintos con copias idénticas de los mismos elementos.

## 10.4. Números aleatorios

Muchos programas de ordenador hacen lo mismo cada vez que se ejecutan, por lo que se dice que son **determinísticos**. Normalmente, el determinismo es algo bueno, ya que esperamos que el mismo cálculo proporcione el mismo resultado. Para ciertas aplicaciones, en cambio, nos gustaría que el ordenador fuese impredecible. Los juegos son un ejemplo obvio, pero hay muchos más.

Hacer un programa verdaderamente **no determinístico** está bastante lejos de ser sencillo, pero hay formas de hacer que al menos parezca no determinístico. Una de ellas es generar números aleatorios y usarlos para determinar la salida del programa. Eiffel proporciona una clase para trabajar con números **pseudoaleatorios**, que no son realmente aleatorios en el sentido matemático del término, pero que para nuestros propósitos lo serán.

Echa un vistazo a la documentación de la clase **STD\_RAND**. Sus métodos más interesantes son **last\_double** y **next**. El valor de retorno de **last\_double** es un valor **DOUBLE** aleatorio entre 0.0 y 1.0 (sin contar el cero). Para generar un nuevo valor pseudoaleatorio se utiliza el método **next**. Como ejemplo, ejecuta este bucle:

```

local
  r: STD_RAND
  i: INTEGER
  x: DOUBLE
do
  from
    create r.make
    i := 1
  until
    i = 10
  loop
    x := r.last_double
    print(x) print("%N")
    r.next
    i := i + 1
  end
end

```

Para generar un valor **DOUBLE** aleatorio entre 0.0 y un límite superior como **alto**, puedes multiplicar **x** por **alto**. ¿Cómo podrías generar un valor aleatorio entre **bajo** y **alto**?

Igualmente, para generar un valor **INTEGER** aleatorio, puedes usar el método **last\_integer** de la clase **STD\_RAND**. Mira la documentación si estás interesado.

## 10.5. Estadísticas

Los números generados por **STD\_RAND** se supone que están distribuidos uniformemente. Si alguna vez has hecho estadísticas, sabrás lo que eso significa. Entre otras cosas, significa que si dividimos el rango de posibles valores en “trozos” de igual tamaño, y contamos el número de veces que un valor aleatorio cae en cada trozo, ese número debe ser aproximadamente igual para cada trozo.

En los próximos apartados, escribiremos programas que generan una secuencia de números aleatorios y comprueban si esa propiedad se cumple.

## 10.6. Array de números aleatorios

El primer paso es generar un gran número de valores aleatorios y almacenarlos luego en un array. Por “gran número”, por supuesto, me refiero a 8. Siempre es buena idea comenzar con un número fácilmente manejable, para ayudarte durante la depuración, y luego aumentarlo posteriormente.

El siguiente método recibe un entero, el tamaño del array. Crea un nuevo array de `DOUBLES`, lo rellena con valores aleatorios, y devuelve una referencia al nuevo array.

```
array_aleatorio(n: INTEGER): ARRAY[DOUBLE] is
  local
    a: ARRAY[DOUBLE]
    r: STD_RAND
    i: INTEGER
  do
    from
      create a.make(1, n)
      create r.make
      i := a.lower
    until
      i > a.upper
    loop
      a.put(r.last_double, i)
      r.next
      i := i + 1
    end
    Result := a
  end
```

El tipo de retorno es `ARRAY[DOUBLE]`, lo que significa que este método devuelve un array de `DOUBLES`. Para comprobar este método, es conveniente hacer un método que imprima el contenido de un array.

```
imprimir_array(a: ARRAY[DOUBLE]) is
  local
    i: INTEGER
  do
    from
      i := a.lower
    until
      i > a.upper
    loop
      print(a @ i) print("%N")
      i := i + 1
    end
  end
```

El siguiente código genera un array y lo imprime:

```

local
  numero_valores: INTEGER
  array: ARRAY[DOUBLE]
do
  numero_valores := 8
  array := array_aleatorio(numero_valores)
  imprimir_array(array)
end

```

En mi máquina la salida es:

```

0.7344558779885422
0.6224282219647016
0.0959142451532917
0.2992298398883563
0.7736458103088713
0.7069110192991597
0.7042440765950522
0.9778395322498520

```

lo que tiene una pinta ciertamente aleatoria. Tus resultados pueden ser distintos a los míos.

Si esos números fuesen verdaderamente aleatorios, deberíamos esperar que la mitad de ellos fueran mayores que 0.5 y la mitad menores que 0.5. De hecho, seis de ellos son mayores que 0.5, lo que está un poco alto.

Si dividimos el rango en cuatro trozos —desde 0.0 hasta 0.25, 0.25 hasta 0.5, 0.5 hasta 0.75, y 0.75 hasta 1.0— esperamos que caigan dos valores en cada trozo. De hecho, obtenemos 1, 1, 4, 2. De nuevo, no es exactamente lo que esperábamos.

¿Significan esos resultados que los valores no son verdaderamente aleatorios? Es difícil de decir. Con tan pocos valores, hay pocas oportunidades de que podamos obtener exactamente lo que esperamos. Pero a medida que el número de valores se incrementa, la salida debería hacerse más predecible.

Para probar esta teoría, escribiremos algunos programas que dividan el rango en trozos y cuenten el número de valores en cada trozo.

## 10.7. Contando

Una buena aproximación a problemas como éste es pensar en métodos simples que sean sencillos de escribir, y que nos puedan ser útiles. Después puedes combinarlos para formar una solución. Por supuesto, no es fácil saber tan pronto qué métodos parece que serán útiles, pero a medida que ganes experiencia lo harás mejor.

Además, no siempre es obvio saber qué tipo de cosas son fáciles de escribir, pero una buena aproximación es mirar subproblemas que se ajusten a un patrón que hayas visto antes.

En el apartado 7.8 vimos un bucle que recorría una cadena y contaba el número de veces que aparecía cierta letra en la cadena. Puedes pensar en ese programa como un ejemplo de un patrón llamado “recorrer y contar”. Los elementos de este patrón son:

- Un conjunto o contenedor que hay que recorrer, como un array o una cadena.
- Una condición que se debe comprobar en cada elemento del contenedor.
- Un contador que mantiene el número de elementos que van pasando el test.

En este caso, tengo un método en mente llamado `en_trozo` que cuenta el número de elementos de un array que caen en un determinado trozo. Los parámetros son el array y dos `DOUBLE`s que especifican los límites inferior y superior del trozo.

```

en_trozo(a: ARRAY[DOUBLE]; inf, sup: DOUBLE): INTEGER is
  local
    contador, i: INTEGER
  do
    from
      contador := 0
      i := a.lower
    until
      i > a.upper
    loop
      if (a @ i) >= inf and (a @ i) < sup then
        contador := contador + 1
      end
      i := i + 1
    end
  Result := contador
end

```

No he sido muy cuidadoso con respecto a lo que ocurre cuando algo resulta ser igual a `inf` o `sup`, así que puedes ver a partir del código que `inf` está dentro del trozo y `sup` está fuera. Esto me previene de contar algún elemento dos veces.

Ahora, para dividir el rango en dos trozos, podemos escribir

```

local
  inf, sup: INTEGER
do
  inf := en_trozo(a, 0.0, 0.5)
  sup := en_trozo(a, 0.5, 1.0)
end

```

Para dividirlo en cuatro trozos:

```

local
  trozo1, trozo2, trozo3, trozo4: INTEGER
do
  trozo1 := en_trozo(a, 0.0, 0.25)
  trozo2 := en_trozo(a, 0.25, 0.5)
  trozo3 := en_trozo(a, 0.5, 0.75)
  trozo4 := en_trozo(a, 0.75, 1.0)
end

```

Puedes intentar este programa usando un número mayor de `numero_valores`. A medida que `numero_valores` crece, ¿se equilibran los números en cada trozo?

## 10.8. Muchos trozos

Por supuesto, cuando crezca el número de trozos, no queremos tener que reescribir el programa, especialmente porque el código se va haciendo grande y repetitivo. Cada vez que te encuentres haciendo algo más de unas pocas veces, deberías buscar una forma de automatizarlo.

Digamos que queremos 8 trozos. La anchura de cada trozo debería ser un octavo del tamaño del rango, que es 0.125. Para contar el número de valores en cada trozo, necesitamos ser capaces de generar los límites de cada trozo automáticamente, y necesitamos tener algún lugar donde almacenar los 8 contadores.

Podemos resolver el primer problema con un bucle:

```
local
  numero_trozos, i: INTEGER
  ancho_trozo, inf, sup: DOUBLE
do
  numero_trozos := 8
  ancho_trozo := 1.0 / numero_trozos

  from
    i := 0
  until
    i >= numero_trozos
  loop
    inf := i * ancho_trozo
    sup := inf + ancho_trozo
    print(inf) print(" a ") print(sup)
    i := i + 1
  end
end
```

Este código utiliza la variable contador `i` multiplicándola por el ancho del trozo, con idea de calcular el límite inferior de cada trozo. La salida de este bucle es:

```
0.0 a 0.125
0.125 a 0.25
0.25 a 0.375
0.375 a 0.5
0.5 a 0.625
0.625 a 0.75
0.75 a 0.875
0.875 a 1.0
```

Puedes confirmar que cada trozo tiene la misma anchura, que no se solapan, y que cubren el rango completo desde 0.0 hasta 1.0.

Ahora sólo necesitamos una manera de almacenar 8 enteros, preferiblemente de forma que podamos usar un índice para acceder a cada uno. Inmediatamente, deberás haber pensado “¡array!”.

Lo que queremos es un array de 8 enteros, que podamos ubicar fuera del bucle; después, dentro del bucle, invocaremos a `en_trozo` y almacenaremos el resultado:

```
local
  numero_trozos, i: INTEGER
  trozos: ARRAY[INTEGER]
  ancho_trozo, inf, sup: DOUBLE
do
  numero_trozos := 8
  create trozos.make(1, 8)
  ancho_trozo := 1.0 / numero_trozos
```



```

    from
      i := 0
    until
      i >= numero_trozos
    loop
      inf := i * ancho_trozo
      sup := inf + ancho_trozo
      -- print(inf) print(" a ") print(sup)
      trozos.put(en_trozo(a, inf, sup), i + 1)
      i := i + 1
    end
  end
end

```

El truco aquí es que estoy usando la variable contador como índice para el array `trozos`, además de usarla para calcular el rango de cada trozo. Observa, en todo caso, que el índice más pequeño del array es 1, no 0 (que es el valor más pequeño que toma `i`), y por consiguiente hay que usar como índice la expresión `i + 1` en lugar de `i` en la sentencia `put`.

Este código funciona. Calculé el número de valores hasta 1000 y dividí el rango en 8 trozos. La salida es:

```

129
109
142
118
131
124
121
126

```

lo que está muy cerca de 125 en cada trozo. Al menos, está lo bastante cerca como para poder creer que el generador de números aleatorios funciona.

## 10.9. Una solución de un sólo paso

Aunque este código funciona, no es tan eficiente como podría ser. Cada vez que se llama a `en_trozo`, éste recorre el array completo. A medida que crece el número de trozos, éste llega a ser un montón de recorridos.

Podría ser mejor hacer una única pasada a través del array, y para cada valor, calcular dentro de qué trozo cae. Entonces podríamos incrementar el contador apropiado.

En el apartado anterior, cogimos un índice, `i`, y lo multiplicamos por el `ancho_trozo` con idea de calcular el límite inferior de un trozo dado. Ahora queremos tomar un valor dentro del rango de 0.0 a 1.0, y calcular el índice del trozo en donde cae.

Ya que este problema es el inverso del problema anterior, podríamos preguntarnos si deberíamos *dividir* por el `ancho_trozo` en lugar de multiplicar. Esa suposición es correcta.

Recuerda que como `ancho_trozo := 1.0 / numero_trozos`, dividir por `ancho_trozo` es lo mismo que multiplicar por `numero_trozos`. Si cogemos un número entre 0.0 y 1.0, y lo multiplicamos por `numero_trozos`, obtenemos un número entre 0.0 y `numero_trozos`. Si redondeamos ese número al siguiente entero más próximo, obtendremos exactamente lo que estábamos buscando — el índice del trozo apropiado.

```

local
  numero_trozos, i, indice: INTEGER
  trozos: ARRAY[INTEGER]
do
  numero_trozos := 8
  create trozos.make(1, 8)

  from
    i := 1
  until
    i > numero_valores
  loop
    indice := ((a @ i) * numero_trozos).floor + 1
    trozos.put((trozos @ indice) + 1, indice)
    i := i + 1
  end
end
end

```

Al índice calculado se le suma 1 por la misma razón que en el caso anterior: el índice más pequeño del array `trozos` es 1, y no 0.

¿Es posible que un cálculo produzca un índice que esté fuera del rango (que sea negativo o mayor que `a.count - 1`)? Si es así, ¿cómo podrías arreglarlo?

Un array como `trozos`, que contiene contadores del número de valores en cada rango, se llama un **histograma**. Como ejercicio, escribe un método llamado `histograma` que reciba un array y un número de trozos como parámetros, y que devuelva un histograma que con el número de trozos dado.

## 10.10. Glosario

**array:** Una colección de valores al que se le da un nombre, donde todos los valores son del mismo tipo, y cada valor se identifica por un índice.

**colección:** Cualquier estructura de datos que contiene un conjunto de ítems o elementos.

**elemento:** Uno de los valores de un array. El operador `@` selecciona elementos de un array.

**índice:** Una variable o valor entero usado para indicar un elemento de un array.

**determinístico:** Un programa que hace lo mismo cada vez que se ejecuta.

**pseudoaleatorio:** Una secuencia de números que parecen ser aleatorios, pero que en realidad son el producto de un cálculo determinístico.

**histograma:** Un array de enteros donde cada entero cuenta el número de valores que caen dentro de un determinado rango.

## Capítulo 11

# Arrays de objetos

### 11.1. Composición

Hasta ahora hemos visto varios ejemplos de composición (la capacidad de combinar características del lenguaje organizándolas de varias formas). Uno de los primeros ejemplos que vimos consistía en usar la llamada a un método como parte de una expresión. Otro ejemplo es la estructura anidada de sentencias: puedes colocar una sentencia `if` dentro de un bucle `loop`, o dentro de otra sentencia `if`, etc.

Habiendo visto este patrón, y habiendo aprendido sobre arrays y objetos, no debería sorprenderte el saber que puedes tener arrays de objetos. De hecho, en Eiffel, todos los valores son objetos de una cierta clase, así que un array de `DOUBLES` es realmente ya un array de objetos (objetos de la clase `DOUBLE`). Además, también puedes tener objetos que contienen arrays (como atributos); puedes tener arrays que contienen a otros arrays; puedes tener objetos que contienen a otros objetos, y así.

En los próximos dos capítulos veremos algunos ejemplos de esas combinaciones, usando objetos `CARTA` como ejemplo.

### 11.2. Objetos `CARTA`

Si no estás familiarizado con los tradicionales juegos de cartas, ahora puede ser un buen momento para buscar una baraja, pues de otra forma este capítulo no tendría mucho sentido. Hay 52 cartas en una baraja francesa, cada una de las cuales pertenece a uno de cuatro palos y uno de 13 valores. Los palos son Picas, Corazones, Diamantes y Tréboles (en orden descendente según el Bridge). Los valores son As, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jota, Reina y Rey. Dependiendo del juego al que estés jugando, el valor de un As puede ser más grande que el de un Rey o más pequeño que el de un 2.

Si queremos definir un nuevo objeto que represente a una carta de la baraja, parece obvio que los atributos deberían ser: **valor** y **palo**. No resulta tan obvio qué tipo de atributos deberían ser. Una posibilidad es `STRING`, de forma que contengan cosas como "`Pica`" para el los palos y "`Reina`" para el valor. Un problema de esta implementación es que puede no resultar fácil comparar cartas para ver cuál tiene el orden o el palo mayor.

Una alternativa es usar enteros para **codificar** los valores y los palos. Por “codificar”, no me refiero a lo que alguna gente piensa, que sería encriptar, o traducir a un código secreto. Lo que los informáticos quieren decir con “codificar” es algo parecido a “definir una correspondencia entre una secuencia de números y aquello que se quiere representar”. Por ejemplo,

Picas	$\mapsto$	4
Corazones	$\mapsto$	3
Diamantes	$\mapsto$	2
Tréboles	$\mapsto$	1

El símbolo  $\mapsto$  es la notación matemática para “se corresponde con”. La característica obvia de esta correspondencia es que los palos se corresponden con enteros ordenados, por lo que podemos comparar palos comparando enteros. La correspondencia para valores es claramente obvia; cada uno de los valores numéricos se corresponden con su valor entero, y para las figuras:

Jota	$\mapsto$	11
Reina	$\mapsto$	12
Rey	$\mapsto$	13

La razón por la que estoy usando una notación matemática para estas correspondencias es que no forman parte del programa Eiffel. Son parte del diseño del programa, pero nunca aparecerán explícitamente en el código. La definición de clase para el tipo **CARTA** tiene el siguiente aspecto:

```
class CARTA

creation
  make,
  make_par

feature
  palo, valor: INTEGER

  make is
    do
      palo := 1
      valor := 1
    end

  make_par(p, v: INTEGER) is
    do
      palo := p
      valor := v
    end
end
```

Como ya es usual, estoy proporcionando dos constructores, uno de los cuales recibe un parámetro por cada atributo, y el otro no recibe parámetros.

Para crear un objeto que represente al tres de tréboles, podríamos usar la sentencia **create**:

```
local
  tres_de_treboles: CARTA
do
  create tres_de_treboles.make_par(1, 3)
end
```

El primer argumento, 1, representa el palo de los tréboles.

### 11.3. El método `imprimir_carta`

Cuando creas una nueva clase, el primer paso normalmente es declarar los atributos y escribir constructores. El segundo paso a menudo es escribir los métodos estándar que todo objeto debería tener, incluido uno que imprima el objeto, y uno o dos que comparen objetos. Comenzaré con `imprimir_carta`.

Con vistas a imprimir objetos `CARTA` de forma que los humanos puedan leerlo fácilmente, queremos transformar los códigos enteros en palabras. Una forma natural de hacer eso es con un array de `STRINGS`. Puedes crear un array de `STRINGS` de la misma forma que creas un array de otros tipos:

```
local
  palos: ARRAY[STRING]
do
  create palos.make(1, 4)
end
```

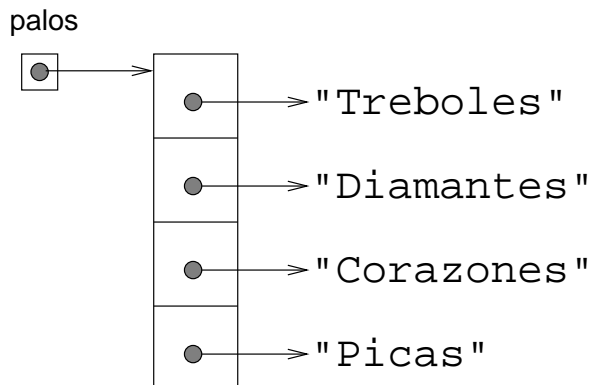
Luego podemos asignar los valores de los elementos del array.

```
palos.put("Tréboles", 1)
palos.put("Diamantes", 2)
palos.put("Corazones", 3)
palos.put("Picas", 4)
```

Crear un array e inicializar los elementos es una operación tan común que Eiffel proporciona una sintaxis especial para ello:

```
local
  palos: ARRAY[STRING]
do
  palos := << "Tréboles", "Diamantes", "Corazones", "Picas" >>
end
```

El efecto de esta sentencia es idéntica al de la llamada a `create` y las cuatro llamadas a `put`, todas juntas. Un diagrama de estado para este array podría tener este aspecto:



Los elementos del array son *referencias* a los `STRINGS`, en lugar de `STRINGS` en sí mismos. Recuerda que esto es cierto para todos los tipos *no expandidos* (como vimos en el capítulo 8). Por ahora, todo lo que necesitamos es otro array de `STRINGS` para decodificar los valores:

```

local
  valores: ARRAY[STRING]
do
  valores := << "As", "2", "3", "4", "5", "6",
               "7", "8", "9", "10", "Jota", "Reina", "Rey" >>
end

```

Usando estos arrays, podemos seleccionar la cadena apropiada usando `palo` y `valor` como índices. En el método `imprimir_carta`,

```

imprimir_carta(c: CARTA) is
  local
    palos, valores: ARRAY[STRING]
  do
    palos := << "Tréboles", "Diamantes", "Corazones", "Picas" >>
    valores := << "As", "2", "3", "4", "5", "6",
                 "7", "8", "9", "10", "Jota", "Reina", "Rey" >>
    print(valores @ (c.valor))
    print(" de ")
    print(palos @ (c.palo))
  end
end

```

La expresión `palos @ (c.palo)` significa “usa el atributo `palo` del objeto `c` como índice para el array llamado `palos`, y selecciona la cadena apropiada”. La salida de este código:

```

local
  carta: CARTA
do
  create carta.make_par(2, 11)
  imprimir_carta(carta)
end

```

es Jota de Diamantes.

## 11.4. El método `misma_carta`

La palabra “mismo” es una de esas cosas que ocurren en el lenguaje natural que parecen perfectamente claras hasta que lo piensas un poco, y entonces te das cuenta de que hay más en ella de lo que esperabas.

Por ejemplo, si yo digo “Juan y yo tenemos el mismo coche”, me estoy refiriendo a que su coche y el mío tienen la misma marca y modelo, pero que son dos coches diferentes. Si yo digo “Juan y yo tenemos la misma madre”, significa que su madre y la mía son una y la misma. Por tanto, lo que se entienda por “misma” es diferente dependiendo del contexto.

Cuando hablas sobre objetos, existe una ambigüedad similar. Por ejemplo, si dos `CARTAs` son la misma, ¿eso significa que contienen los mismos datos (valor y palo), o que en realidad son el mismo objeto `CARTA`?

Para ver si dos referencias apuntan al mismo objeto, podemos usar el operador `=`. Por ejemplo:

```

local
  carta1, carta2: CARTA
do

```

```

create carta1.make_par(2, 11)
carta2 := carta1

if carta1 = carta2 then
    print("carta1 y carta2 son el mismo objeto.%N")
end
end

```

Este tipo de igualdad se denomina **igualdad superficial** porque sólo compara las referencias, no el contenido de los objetos.

Para comparar el contenido de los objetos —**igualdad profunda**— es común escribir un método con un nombre como `misma_carta`.

```

misma_carta(c1, c2: CARTA): BOOLEAN is
do
    Result := c1.palo = c2.palo and c1.valor = c2.valor
end

```

Ahora si creamos dos objetos diferentes que contienen los mismos datos, podemos usar `misma_carta` para ver si representan la misma carta:

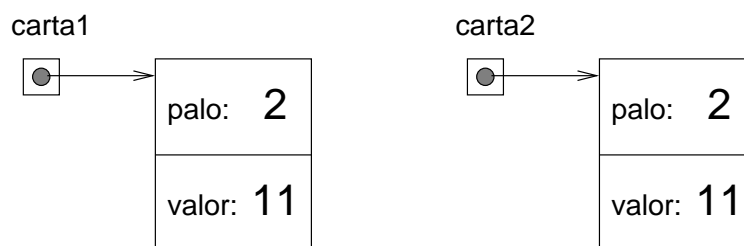
```

local
    carta1, carta2: CARTA
do
    create carta1.make_par(2, 11)
    create carta2.make_par(2, 11)

    if misma_carta(carta1, carta2) then
        print("carta1 y carta2 son la misma carta.%N")
    end
end
end

```

En este caso, `carta1` y `carta2` son dos objetos diferentes que contienen los mismos datos



por lo que la condición es verdadera. ¿Cómo es el diagrama de estado cuando `carta1 = carta2`?

En el apartado 7.11 dije que nunca deberías usar el operador `=` sobre `STRING`s porque no hace lo que tú te esperas. En lugar de comparar los contenidos de las cadenas (igualdad profunda), comprueba si las dos cadenas son el mismo objeto (igualdad superficial).

## 11.5. El método `comparar_carta`

Para los tipos más usuales (`INTEGER`, `DOUBLE`, `STRING`, etc.), existen operadores relacionales que comparan valores y determinan cuándo uno es más grande o más pequeño que otro. Esos

operadores (< y > y los demás) no funcionan sobre cualquier objeto. Para las CARTAs tenemos que escribir nuestro propio método de comparación, al que vamos a llamar `comparar_carta`. Más tarde, usaremos este método para ordenar una baraja de cartas.

Algunos conjuntos están completamente ordenados, lo que significa que puedes comparar cualesquiera dos elementos y decir cuál es más grande. Por ejemplo, los enteros y los reales están totalmente ordenados. Algunos conjuntos no están ordenados, lo que significa que no existe ninguna forma de decir que un elemento es mayor que otro. Por ejemplo, las frutas no están ordenadas, y por eso no podemos comparar manzanas y naranjas. En Eiffel, el tipo `BOOLEAN` no está ordenado; no podemos decir que `True` es mayor que `False`.

El conjunto de las cartas está parcialmente ordenado, lo que significa que a veces podemos comparar cartas y a veces no. Por ejemplo, yo sé que el 3 de Tréboles es mayor que el 2 de Tréboles, y que el 3 de Diamantes es mayor que el 3 de Tréboles. Pero, ¿qué es mejor, el 3 de Tréboles o el 2 de Diamantes? Uno tiene un valor más alto, pero el otro tiene un palo mayor.

Con idea de hacer las cartas comparables, tenemos que decidir qué es más importante, el valor o el palo. Para ser honesto, la elección es completamente arbitraria. Por tener que elegir, diré que el palo es más importante, porque cuando cuando compras una baraja de cartas nueva, ésta viene ordenada con todos los Tréboles juntos, seguidos de todos los Diamantes, y así.

Con eso decidido, podemos escribir `comparar_carta`. Recibirá dos CARTAs como parámetros y devolverá 1 si la primera carta gana, -1 si la segunda carta gana, y 0 si hay empate. A veces confunde mantener esos valores de retorno, pero son casi un estándar para los métodos de comparación.

Primero comparamos los palos:

```
if c1.palo > c2.palo then
  Result := 1
elseif c1.palo < c2.palo then
  Result := -1
else
```

Si ninguna de las condiciones se cumple, entonces los palos son iguales, y tenemos que comparar valores:

```
if c1.palo > c2.palo then
  Result := 1
elseif c1.palo < c2.palo then
  Result := -1
else
  if c1.valor > c2.valor then
    Result := 1
  elseif c1.valor < c2.valor then
    Result := -1
  else
    Result := 0
  end
end
```

Si ninguna de esas condiciones se cumple, los valores deben ser iguales, así que devolvemos 0. En este orden, los ases son más pequeños que los doses (2s).

Como ejercicio, arréglalo de forma que los ases valgan más que los Reyes, y encapsula este código en un método.

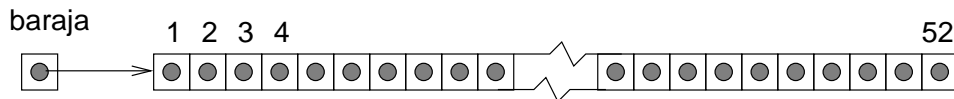


## 11.6. Arrays de cartas

La razón por la que escogí las **CARTAS** como los objetos para este capítulo fue que hay un uso obvio para un array de cartas —una baraja—. He aquí un poco de código que crea una nueva baraja de 52 cartas:

```
local
  baraja: ARRAY[CARTA]
do
  create baraja.make(1, 52)
end
```

Este es el diagrama de estado para este objeto:



Lo importante aquí es que el array contiene sólo *referencias* a objetos; no contiene ningún objeto **CARTA**. Los valores de los elementos del array están inicializados a **Void**. Puedes acceder a los elementos del array de la manera usual:

```
if baraja @ 3 = Void then
  print(";Todavía no hay cartas!%N")
end
```

Pero si intentas acceder a los atributos de **CARTAS** que no existen, obtendrás una excepción y el mensaje de error "Target is Void".

```
(baraja @ 2).valor      -- Target is Void.
```

No obstante, esa es la sintaxis correcta para acceder al valor de la segunda carta de la baraja. Este es otro ejemplo de composición, la combinación de la sintaxis para acceder a un elemento del array y a un atributo de un objeto.

La forma más sencilla de poblar la baraja con objetos **CARTA** es escribir un bucle anidado:

```
local
  indice, palo, valor: INTEGER
  baraja: ARRAY[CARTA]
do
  create baraja.make(1, 52)
  indice := 1

  from
    palo := 1
  until
    palo > 4
  loop
    from
      valor := 1
    until
      valor > 13
```

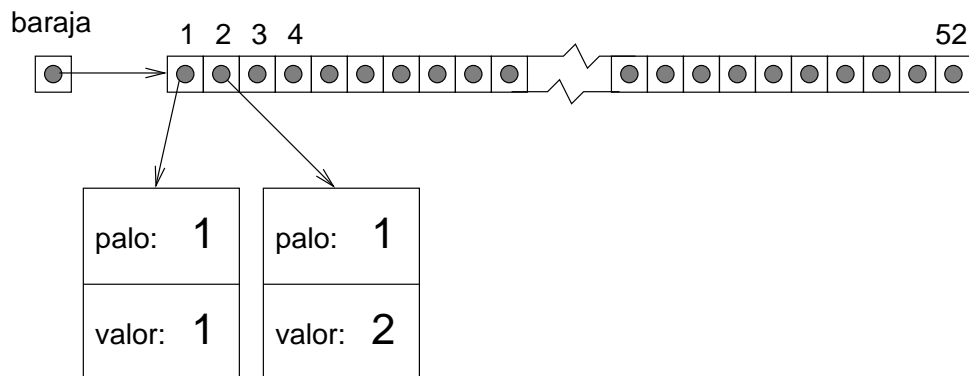
```

    loop
      baraja.put(create {CARTA}.make(palo, valor), indice)
      indice := indice + 1
      valor := valor + 1
    end
    palo := palo + 1
  end
end

```

El bucle más externo enumera los palos, desde 1 hasta 4. Para cada palo, el bucle interno enumera los valores, desde 1 hasta 13. Como el bucle externo se ejecuta 4 veces, y el bucle interno se ejecuta 13 veces, el número total de veces que se ejecuta el cuerpo es 52 (13 por 4).

He usado la variable `indice` para llevar la cuenta de dónde debe ir la siguiente carta en la baraja. El siguiente diagrama de estado muestra el aspecto de la baraja después de haber situado las primeras dos cartas:



Como ejercicio, encapsula este código construye-barajas en un método llamado `crear_baraja` que no reciba parámetros y que devuelva un array totalmente relleno de `CARTAS`.

## 11.7. El método `imprimir_baraja`

Siempre que trabajes con arrays, será conveniente tener un método que imprima el contenido del array. Hemos visto el patrón para recorrer un array varias veces, por lo que el siguiente método te debería resultar familiar:

```

imprimir_baraja(baraja: ARRAY[CARTA]) is
  local
    i: INTEGER
  do
    from
      i := baraja.lower
    until
      i > baraja.upper
    loop
      imprimir_carta(baraja @ i)
      i := i + 1
    end
  end
end

```

Como `baraja` tiene un tipo `ARRAY[CARTA]`, cada elemento de `baraja` tiene el tipo `CARTA`. Por tanto, `baraja @ i` es un argumento válido para `imprimir_carta`.

## 11.8. Búsqueda

El siguiente método que quiero escribir es `buscar_carta`, que busca a través de un array de `CARTAs` para ver si contiene una carta determinada. Puede no ser obvio el por qué este método podría ser útil, pero me da la ocasión de demostrar dos formas de buscar cosas, una búsqueda `lineal` y una búsqueda `binaria`.

La búsqueda lineal es la más obvia de las dos; implica recorrer la baraja y comparar cada carta con la que estamos buscando. Si la encontramos devolvemos el índice donde se encuentra la carta. Si no está en la baraja, devolvemos 0.

```

buscar_carta(baraja: ARRAY[CARTA]; buscada: CARTA): INTEGER is
  local
    i: INTEGER
  do
    from
      i := baraja.lower
      Result := 0
    until
      i > baraja.upper
    loop
      if misma_carta(baraja @ i, buscada) then
        Result := i
        i := baraja.upper
      end
      i := i + 1
    end
  end
end

```

El método finaliza tan pronto descubre la carta buscada, lo que significa que no tenemos que recorrer toda la baraja si encontramos la carta que estábamos buscando. Para lograr este efecto, usamos un pequeño truco: cuando localizamos la carta buscada, asignamos a la variable `i` el índice final del array con la sentencia `i := baraja.upper` (que ya sabemos que es 52). Luego, esa misma variable se incrementa como en cada iteración mediante la sentencia `i := i + 1`, por lo que `i` ya tiene un valor mayor que `baraja.upper`, y por tanto se cumple la condición `i > baraja.upper`, que es precisamente la condición de salida del bucle.

Si el bucle finaliza sin haber encontrado la carta buscada, entonces la variable especial `Result` no cambia su valor (que antes de comenzar el bucle era 0), y por tanto el método devuelve 0.

Si las cartas en la baraja no están en orden, no hay forma de acelerar la búsqueda. Tenemos que mirar cada carta, pues de otro modo no habría manera de asegurarse de que la carta que queremos no está en la baraja.

Pero cuando buscas una palabra en un diccionario, no haces la búsqueda de manera lineal palabra por palabra. La razón es que las palabras están en orden alfabético. Como resultado, probablemente uses un algoritmo similar a una búsqueda binaria:

1. Empieza en algún lugar más o menos a la mitad del diccionario.
2. Elige una palabra de la página y compárala con la palabra que estás buscando.
3. Si encuentras la palabra que estás buscando, para.
4. Si la palabra que estás buscando viene después de la palabra situada en la página, pásate a algún lugar posterior en el diccionario y vuelve al paso 2.

5. Si la palabra que estás buscando viene antes de la palabra situada en la página, pásate a algún lugar anterior en el diccionario y vuelve al paso 2.

Si llegas a un punto en el que hay dos palabras adyacentes en la página y tu palabra debería aparecer entre ellas, puedes concluir que tu palabra no está en el diccionario. La única alternativa es que tu palabra está en otro sitio, pero eso contradice nuestra suposición de que las palabras están en orden alfabético.

En el caso de una baraja de cartas, si sabemos que las cartas están ordenadas, podemos escribir una versión de `buscar_carta` mucho más rápida. La mejor forma de escribir una búsqueda binaria es con un método recursivo. Esto es así porque la bisección es naturalmente recursiva.

El truco está en escribir un método llamado `buscar_binaria` que reciba dos índices como parámetros, `inf` y `sup`, los cuales indican el segmento del array en el que se debe buscar (incluidos `inf` y `sup`).

1. Para buscar en el array, escoge un índice entre `inf` y `sup` (llámalo `mitad`) y compara la carta situada allí con la carta que estas buscando.
2. Si la has encontrado, para.
3. Si la carta que está en `mitad` es mayor que tu carta, busca en el rango situado entre `inf` y `mitad-1`.
4. Si la carta que está en `mitad` es menor que tu carta, busca en el rango situado entre `mitad+1` y `sup`.

Los pasos 3 y 4 se parecen sospechosamente a llamadas recursivas. Este es el aspecto que tiene todo esto traducido a código Eiffel:

```

buscar_binaria(baraja: ARRAY[CARTA]; buscada: CARTA;
               inf, sup: INTEGER): INTEGER is
  local
    mitad, comp: INTEGER
  do
    mitad := (inf + sup) // 2
    comp := comparar_carta(baraja @ mitad, buscada)
    if comp = 0 then
      Result := mitad
    elseif comp > 0 then
      Result := buscar_binaria(baraja, carta, inf, mitad-1)
    else
      Result := buscar_binaria(baraja, carta, mitad+1, sup)
    end
  end
end

```

En lugar de llamar a `comparar_carta` tres veces, la llamo una vez y almaceno el resultado.

Aunque este código contiene el núcleo de una búsqueda binaria, todavía falta un pedazo. Tal y como está escrita ahora, si la carta no está en la baraja, se llamará recursivamente a sí misma para siempre. Necesitamos una manera de detectar esta condición y tratarla adecuadamente (devolviendo 0).

La forma más sencilla de decir que tu carta no está en la baraja se da cuando *no* hay cartas en la baraja, que es el caso en el que `sup` es menor que `inf`. Bueno, todavía hay cartas en la baraja, por supuesto, pero lo que quiero decir es que no hay cartas en el segmento de la baraja indicado por `inf` y `sup`.

Con esa línea añadida, e incluyendo lo demás en una rama de una nueva sentencia `if`, el método funciona correctamente:

```

buscar_binaria(baraja: ARRAY[CARTA]; buscada: CARTA;
               inf, sup: INTEGER): INTEGER is
  local
    mitad, comp: INTEGER
  do
    print(inf) print(", ") print(sup) print("%N")
    if sup < inf then
      Result := 0
    else
      mitad := (inf + sup) // 2
      comp := comparar_carta(baraja @ mitad, buscada)
      if comp = 0 then
        Result := mitad
      elseif comp > 0 then
        Result := buscar_binaria(baraja, carta, inf, mitad-1)
      else
        Result := buscar_binaria(baraja, carta, mitad+1, sup)
      end
    end
  end
end

```

He añadido una sentencia de impresión al principio de forma que pueda ver la secuencia de llamadas recursivas y convencerme a mí mismo de que se alcanza finalmente el caso base. Lo pruebo con el siguiente código:

```

local
  carta1: CARTA
  baraja: ARRAY[CARTA]
do
  -- Crear e inicializar baraja.
  create carta1.make_par(1, 11)
  print(buscar_binaria(baraja, carta1, 0, 51))
end

```

Y obtuve el siguiente resultado:

```

0, 51
0, 24
13, 24
19, 24
22, 24
23

```

Luego creé una carta que no está en la baraja (el 15 de Corazones), e intenté encontrarla. Obtuve lo siguiente:

```

0, 51
0, 24
13, 24
13, 17

```

```

13, 14
13, 12
0

```

Estos tests no demuestran que el programa sea correcto. De hecho, ninguna cantidad de tests pueden demostrar que un programa es correcto. Por otra parte, mirando unos cuantos casos y examinando el código, puedes convencerte a ti mismo.

El número de llamadas recursivas es ciertamente pequeño, por lo general 6 ó 7. Esto significa que sólo tenemos que llamar a `comparar_carta` 6 ó 7 veces, comparadas con las hasta 52 veces de la búsqueda lineal. En general, la búsqueda binaria es mucho más rápida que la lineal, especialmente para arrays grandes.

Dos errores comunes en programas recursivos son olvidar incluir un caso base y escribir la llamada recursiva de manera que nunca se alcance el caso base. Ambos errores causarán una recursión infinita, lo que provocará que Eiffel produzca un error por desbordamiento de la pila.

## 11.9. Barajas y sub-barajas

Observando la interfaz de `buscar_binaria`

```

buscar_binaria(baraja: ARRAY[CARTA]; buscada: CARTA;
               inf, sup: INTEGER): INTEGER is

```

podría tener sentido tratar tres de los parámetros, `baraja`, `inf` y `sup`, como un único parámetro que especifique una **sub-baraja**.

Este tipo de cosas son bastante comunes, y a veces se las llama **parámetro abstracto**. Lo que quiero decir con “abstracto” es que es algo que no forma parte literal del texto del programa, pero que describe el funcionamiento del programa a un nivel superior.

Por ejemplo, cuando llamas a un método y pasas un array y los límites `inf` y `sup`, nada previene al método llamado de acceder a partes del array que están fuera de los límites. Por lo tanto, no estás literalmente enviando un subconjunto de la baraja; en realidad estás enviando la baraja completa. Pero siempre y cuando el receptor cumpla las reglas del juego, tiene sentido pensar en él (el array y los límites), de forma abstracta, como en una sub-baraja.

Hay otro ejemplo de este tipo de abstracción del que has tenido noticia en el apartado 9.8, cuando me referí a una estructura de datos “vacía”. La razón por la que puse “vacía” entre comillas fue la de sugerir que no era literalmente preciso. Todas las variables tienen valores en todo momento. Cuando las creas, toman valores por defecto. Por tanto, no existe eso de “objeto vacío”.

Pero si el programa garantiza que el valor actual de una variable nunca va a leerse antes de que se almacenen, entonces el valor actual es irrelevante. De manera abstracta, tiene sentido pensar que esa variable está “vacía”.

Esta forma de pensar, en la que un programa llega a tener un significado más allá de lo que está codificado literalmente, es una parte muy importante de la manera de pensar de un informático. A veces, se utiliza la palabra “abstracto” demasiado a menudo y en demasiados contextos de manera que llega a perder su significado. No obstante, la abstracción es una idea central en Informática (así como en muchos otros campos).

Una definición más general de “abstracción” es “El proceso de modelar un sistema complejo con una descripción simplificada para suprimir los detalles innecesarios mientras se capta el comportamiento importante”.

## 11.10. Glosario

**codificar:** Representar un conjunto de valores usando otro conjunto de valores, construyendo una correspondencia entre ellos.

**igualdad superficial:** Igualdad entre referencias. Dos referencias que apuntan al mismo objeto.

**igualdad profunda:** Igualdad entre valores. Dos referencias que apuntan a objetos que tienen el mismo valor.

**parámetro abstracto:** Un conjunto de parámetros que actúan conjuntamente como un único parámetro.

**abstracción:** El proceso de interpretar un programa (o cualquier otra cosa) a un nivel superior al que está representado literalmente por el código.

## Capítulo 12

# Objetos de Arrays

En el capítulo anterior, trabajamos con un array de objetos, pero también mencioné que es posible tener un objeto que contenga a un array como atributo. En este capítulo voy a crear un nuevo objeto, llamado **BARAJA**, que contiene un array de **CARTAS** como atributo.

La definición de clase se parece a esto:

```
class BARAJA

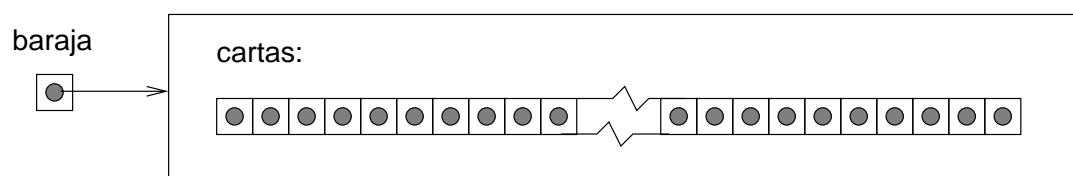
  creation make

  feature
    cartas: ARRAY[CARTA]

    make(n: INTEGER) is
      do
        create cartas.make(1, n)
      end
    end

end
```

El nombre del atributo es **cartas** para ayudar a distinguir el objeto **BARAJA** del array de **CARTAS** que contiene. Este es un diagrama de estado que muestra el aspecto de un objeto **BARAJA** sin cartas creadas:



Como es usual, el constructor inicializa el atributo, pero en este caso utiliza la sentencia **create** para crear el array de cartas. No crea ninguna carta, sin embargo. Para eso tenemos que escribir otro constructor que cree una baraja estándar de 52 cartas y la pueble con objetos **CARTA**.

```
make_estandar is
  local
    indice, palo, valor: INTEGER
  do
```



```

        create cartas.make(1, 52)
    from
        indice := 0
        palo := 1
    until
        palo > 4
    loop
        from
            valor := 1
        until
            valor > 13
        loop
            cartas.put(create {CARTA}.make_par(palo, valor), indice)
            indice := indice + 1
            valor := valor + 1
        end
        palo := palo + 1
    end
end
end

```

Observa lo parecido que es este método con `crear_baraja`. Para que sea un constructor, tenemos que poner su nombre en la cláusula `creation` de la clase `BARAJA`. De esta forma, podremos crear una nueva baraja con la sintaxis habitual:

```

local
    bar: BARAJA
do
    create bar.make_estandar
end

```

Ahora que tenemos una clase `BARAJA`, tiene sentido meter todos los métodos que pertenecen a `BARAJA`s dentro de la definición de la clase `BARAJA`. Observando los métodos que hemos escrito hasta ahora, un candidato obvio es `imprimir_baraja` (apartado 11.7). Este sería su aspecto, reescrito para trabajar con un objeto `BARAJA`:

```

imprimir_baraja(bar: BARAJA) is
    local
        i: INTEGER
    do
        from
            i := bar.cartas.lower
        until
            i > bar.cartas.upper
        loop
            imprimir_carta(bar.cartas @ i)
            i := i + 1
        end
    end
end

```

La cosa más obvia que hemos tenido que cambiar es el tipo del parámetro, de `ARRAY[CARTA]` a `BARAJA`. El segundo cambio es que ya no podemos usar `baraja.lower` ni `baraja.upper` para saber los límites inferior y superior del array, porque `bar` ahora es un objeto `BARAJA`, no un array. Contiene un array, pero no es, en sí mismo, un array. Por tanto, tenemos que escribir

`bar.cartas.lower` y `bar.cartas.upper`, es decir, extraer el array del objeto `BARAJA` y obtener sus límites.

Por la misma razón, tenemos que usar `bar.cartas @ i` para acceder a un elemento del array, en lugar de usar sólo `bar @ i`. Finalmente, tenemos que cambiar un detalle que en el código no resulta claro, pero que es vital para que el programa funcione: tenemos que incluir la definición del método `imprimir_carta` dentro de la definición de la clase `BARAJA`. Esta no es la solución más adecuada, pero todo cambiará más adelante.

Para algunos de los demás métodos, no está claro si deberían ser incluidos en la clase `CARTA` o en la clase `BARAJA`. Por ejemplo, `buscar_carta` recibe una `CARTA` y una `BARAJA` como argumentos; podrías razonablemente meterlo en cualquier clase. Como ejercicio, mete `buscar_carta` dentro de la clase `BARAJA` y reescríbelo de forma que el primer parámetro sea un objeto `BARAJA` en lugar de un array de `CARTAS`.

## 12.1. Barajar

Para muchos juegos de cartas se necesita la capacidad de barajar las cartas; esto es, situar las cartas según un orden aleatorio. En el apartado 10.4 vimos cómo generar números aleatorios, pero ahora no está claro cómo usarlos para barajar un mazo de cartas.

Una posibilidad es modelar la forma en la que los humanos barajan, que normalmente consiste en dividir la baraja en dos y luego formar la baraja eligiendo alternativamente de cada mitad. Como los humanos no barajan perfectamente, después de unas 7 iteraciones el orden de la baraja será bastante aleatorio. Pero un programa de ordenador tiene la molesta propiedad de barajar perfectamente cada vez, lo que verdaderamente no es muy aleatorio. De hecho, después de haber barajado perfectamente 8 veces, podrías acabar con la baraja ordenada de la misma forma en como empezó. Para un debate sobre esta afirmación, visita <http://www.wiskit.com/marilyn/craig.html> o busca en la web las palabras “barajado perfecto” o “perfect shuffling”.

Un mejor algoritmo de barajado es recorrer la baraja una carta cada vez, y en cada iteración elegir dos cartas e intercambiarlas.

He aquí un boceto de cómo funciona este algoritmo. Para esbozar el programa, estoy usando una combinación de sentencias Eiffel y palabras en castellano, lo que a veces se llama **pseudocódigo**:

```

local
  i: INTEGER
do
  from
    i := bar.cartas.lower
  until
    i > bar.cartas.upper
  loop
    -- elegir un número aleatorio entre i y bar.cartas.upper
    -- intercambiar la i-ésima carta y la carta elegida aleatoriamente
    i := i + 1
  end
end
end

```

Lo bueno de usar pseudocódigo es que a menudo aclara qué métodos vas a necesitar. En este caso, necesitamos algo como `entero_aleatorio`, que escoja un número aleatorio entre los parámetros `inf` y `sup`, y `cambiar_cartas` que reciba dos índices e intercambie las cartas de las posiciones indicadas.

Probablemente puedas hacerte una idea de cómo escribir `entero_aleatorio` mirando en el apartado 10.4, aunque tendrás que ser cuidadoso con los índices generados que puedan quedar fuera del rango.

También puedes idear `cambiar_cartas` tú mismo. El único truco es decidir si quieres intercambiar sólo las referencias a las cartas o el contenido de las cartas. ¿Importa cuál de las dos opciones elijas? ¿Cuál es más rápida?

Dejaré la restante implementación de estos métodos como ejercicio para el lector.

## 12.2. Ordenar

Ahora que hemos liado la baraja, necesitamos una manera de volverla a ordenar. Irónicamente, existe un algoritmo de ordenación que es muy similar al algoritmo de barajado. Este algoritmo a menudo se llama **ordenación por selección** porque funciona recorriendo el array repetidamente y seleccionando la carta restante más pequeña cada vez.

Durante la primera iteración buscamos la carta más pequeña y la intercambiamos con la situada en la primera posición. Durante la iteración  $i$ -ésima, buscamos la carta más pequeña a la derecha de  $i$  y la intercambiamos con la carta  $i$ -ésima.

Este es un pseudocódigo para la ordenación por selección:

```
ordenar_baraja(bar: BARAJA): BARAJA is
  local
    i: INTEGER
  do
    from
      i := bar.cartas.lower
    until
      i > bar.cartas.upper
    loop
      -- buscar la carta menor a la derecha de i
      -- intercambiar la carta i-ésima y la carta menor
      i := i + 1
    end
  end
end
```

De nuevo, el pseudocódigo ayuda al diseño de los **métodos auxiliares**. En este caso, podemos usar `cambiar_cartas` otra vez, por lo que solamente necesitamos uno nuevo, llamado `buscar_carta_menor`, que recibe un array de cartas y el índice por donde debe empezar a buscar.

Una vez más, voy a dejar el resto de la implementación al lector.

## 12.3. Sub-barajas

¿Cómo podemos representar una mano, o algún otro subconjunto de una bajara completa? Una buena elección es hacer un objeto `BARAJA` que tenga menos de 52 cartas.

Podríamos querer un método, `sub_baraja`, que reciba un array de cartas y un rango de índices, y que devuelva un nuevo array de cartas que contenga el subconjunto de la baraja que se haya especificado:

```
sub_baraja(bar: BARAJA; inf, sup: INTEGER): BARAJA is
  local
```

```

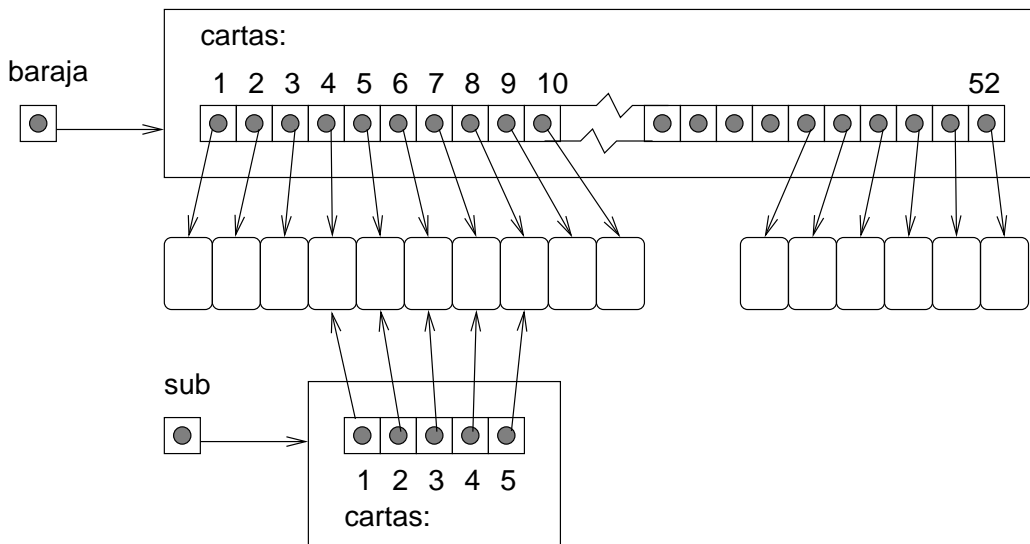
sub: BARAJA
i: INTEGER
do
  create sub.make(sup - inf + 1)
  from
    i := sub.cartas.lower
  until
    i > sub.cartas.upper
  loop
    sub.cartas.put(bar.cartas @ (inf + i), i)
    i := i + 1
  end
  Result := sub
end

```

El tamaño de la sub-baraja es  $\text{sup}-\text{inf}+1$  porque la carta inferior y la carta superior están incluidas. Este tipo de cálculo puede ser confuso, y dar lugar a errores del tipo “pasarse por uno”. Hacer un dibujo es normalmente la mejor manera de evitarlos.

Como utilizamos el constructor `make` en lugar de `make_estandar`, sólo se crea el array pero no las cartas que lo forman. Dentro del bucle, se puebla la sub-baraja con copias de las referencias incluidas en la baraja.

El siguiente es un diagrama de estado de una sub-baraja recién creada con los parámetros  $\text{inf}=3$  y  $\text{sup}=7$ . El resultado es una mano con 5 cartas compartidas con la baraja original; es decir, se producen alias.



Ya sugerí que los alias no son normalmente una buena idea, porque los cambios en una sub-baraja se reflejarían en otras, lo que no es el comportamiento que podrías esperar de cartas y barajas auténticas. Pero si los objetos en cuestión no cambian, entonces los alias son una elección razonable. En este caso, no existe un motivo probable por el que se tengan que cambiar el palo o el valor de una carta. En su lugar, creamos cada carta una sola vez y luego la trataremos como un objeto inmutable. Por tanto, para las CARTAS los alias son una elección razonable.

Como ejercicio, escribe una versión de `buscar_binaria` que reciba una sub-baraja como argumento, en lugar de una baraja y un rango de índices. ¿Qué versión es más propensa a errores? ¿Qué versión crees que es más eficiente?

## 12.4. Barajar y dar

En el apartado 12.1 escribí un pseudocódigo para un algoritmo de barajado. Suponiendo que tenemos un método llamado `barajar_baraja` que recibe una baraja como argumento y le hace un barajado, podemos crear y barajar una baraja:

```
local
  bar: BARAJA
do
  create bar.make_estandar
  barajar_baraja(bar)
end
```

Después, para dar varias manos, podemos usar `sub_baraja`:

```
local
  mano1, mano2, monton: BARAJA
do
  mano1 := sub_baraja(bar, 1, 5)
  mano2 := sub_baraja(bar, 6, 10)
  monton := sub_baraja(bar, 11, 52)
end
```

Este código pone las primeras 5 cartas en una mano, las siguientes 5 cartas en otra, y el resto en el montón.

Cuando piensas en dar cartas, ¿crees que deberíamos dar una carta cada vez a cada jugador por turnos, como es tradicional en juegos reales de cartas? Yo lo pensé, pero luego me dí cuenta de que no es necesario para un programa de ordenador. El convenio de dar las cartas por turnos, una cada vez, intenta mitigar los efectos de un barajado imperfecto y que le resulte más difícil hacer trampas al que da la mano. Ninguna de estas cosas importan en un ordenador.

Este ejemplo es un útil recordatorio de uno de los peligros de las metáforas en ingeniería: a veces imponemos restricciones innecesarias a los ordenadores, o esperamos capacidades que en realidad son carencias, porque inconscientemente ampliamos la metáfora más allá de su punto de ruptura. Cuidado con las analogías engañosas.

## 12.5. Ordenación por mezcla

En el apartado 12.2, vimos un algoritmo sencillo de ordenación que no resulta ser muy eficiente. Para ordenar  $n$  elementos, hay que recorrer el array  $n$  veces, y cada recorrido supone una cantidad de tiempo que es proporcional a  $n$ . El tiempo total, por tanto, es proporcional a  $n^2$ .

En esta sección, esbozaré un algoritmo más eficiente llamado **ordenación por mezcla**. Para ordenar  $n$  elementos, la ordenación por mezcla consume un tiempo proporcional a  $n \log n$ . Esto puede no parecer muy impresionante, pero a medida que  $n$  va creciendo, la diferencia entre  $n^2$  y  $n \log n$  puede llegar a ser enorme. Prueba con unos cuantos valores de  $n$  y míralo.

La idea básica detrás de la ordenación por mezcla es la siguiente: si tienes dos sub-barajas, cada una de las cuales ya ha sido ordenada, es fácil (y rápido) mezclarlas en una sola baraja ordenada. Compruébalo con una baraja de cartas:

1. Forma dos sub-barajas de aproximadamente 10 cartas cada una, y ordénalas de forma que las cartas queden boca arriba, con las más pequeñas en la parte superior. Coloca ambos montones delante de ti.

2. Compara las cartas superiores de cada montón y escoge la menor de las dos. Cógela y añádela a la baraja mezclada.
3. Repite el paso dos hasta que uno de los montones quede vacío. Luego coge las cartas restantes y añádelas a la baraja mezclada.

El resultado debería ser una única baraja ordenada. Este es el aspecto en pseudocódigo:

```
mezcla(b1, b2: BARAJA): BARAJA is
  local
    res: BARAJA
    i, j: INTEGER
  do
    -- crea una nueva baraja lo bastante grande como para que quepan todas las cartas
    create res.make(b1.cartas.count + b2.cartas.count)
    -- usa el índice i para llevar la pista de dónde estamos en
    -- el primer montón, y el índice j para el segundo montón
    i := b1.cartas.lower
    j := b2.cartas.lower

    -- el índice k recorre la baraja resultado
    from
      k := res.cartas.lower
    until
      k > res.cartas.upper
    loop
      -- si b1 está vacío, b2 gana; si b2 está vacío, b1 gana
      -- en otro caso, compara las dos cartas
      -- añade el ganador a la nueva baraja
      k := k + 1
    end
    Result := res
  end
```

La mejor forma de comprobar `mezcla` es crear y barajar una baraja, usar `sub_baraja` para formar dos (pequeñas) manos y después usar el método de ordenación ya conocido para ordenar las dos mitades. Después puedes pasar las dos mitades a `mezcla` para ver si funciona.

Si logras hacer que funcione, intenta una implementación sencilla de `ordenar_mezcla`:

```
ordenar_mezcla(bar: BARAJA): BARAJA is
  do
    -- busca el punto medio de la baraja
    -- divide la baraja en dos sub-barajas
    -- ordena las sub-bajas usando ordenar_baraja
    -- mezcla las dos mitades y devuelve el resultado
  end
```

Después, si consigues que funcione, ¡empieza la verdadera diversión! Lo mágico de `ordenar_mezcla` es que es recursivo. En el momento en que ordenas las sub-barajas, ¿por qué debes usar la vieja y lenta versión de `ordenar_baraja`? ¿Por qué no usar el estupendo y nuevo `ordenar_mezcla` que estás en proceso de escribir?

No sólo es una buena idea: es *necesario* para poder alcanzar la ventaja en prestaciones que prometí. Para hacer que funcione, no obstante, tienes que añadir un caso base de forma que no caiga en

una recursión infinita. Un caso base sencillo es una sub-baraja de 0 ó 1 cartas. Si `ordenar_mezcla` recibe esa pequeña sub-baraja, puede devolverla inalterada, puesto que ya está ordenada.

La versión recursiva de `ordenar_mezcla` debería parecerse a esto:

```
ordenar_mezcla(bar: BARAJA): BARAJA is
do
    -- si la baraja tiene 0 ó 1 cartas, devuélvela
    -- busca el punto medio de la baraja
    -- divide la baraja en dos sub-barajas
    -- ordena las sub-barajas usando ordenar_mezcla
    -- mezcla las dos mitades y devuelve el resultado
end
```

Como es usual, hay dos maneras de pensar en programas recursivos: puedes pensar en seguir el flujo completo de ejecución, o puedes dar el “salto de fe”. He construido este ejemplo deliberadamente para que des el salto de fe.

Cuando estuviste usando `ordenar_baraja` para ordenar las sub-barajas, no te sentiste obligado a seguir el flujo de ejecución, ¿verdad? Simplemente supusiste que el método `ordenar_baraja` podría funcionar porque ya lo habías depurado. Bien, todo lo que hiciste para hacer que `ordenar_mezcla` fuese recursivo fue sustituir un algoritmo de ordenación por otro. No hay motivos para tener que leer el programa de forma diferente.

Bueno, en realidad tuviste que pensar un poco para obtener el caso base adecuado y asegurarte de que se alcanzará en un momento dado, pero por lo demás, escribir la versión recursiva no debería ser problema. ¡Buena suerte!

## 12.6. Glosario

**pseudocódigo:** Una manera de diseñar programas escribiendo bocetos en una combinación de castellano y Eiffel.

**método auxiliar:** A menudo un método pequeño que no hace nada enormemente útil por sí mismo, sino que ayuda a otro método más útil.

## Capítulo 13

# Programación orientada a objetos

### 13.1. Lenguajes y estilos de programación

Existen muchos lenguajes de programación en el mundo, y casi tantos como estilos de programación (a veces llamados paradigmas). Los tres estilos que han aparecido en este libro son el imperativo (o procedimental), el funcional y el orientado a objetos. Aunque Eiffel a menudo es considerado un lenguaje orientado a objetos, es posible escribir programas Eiffel en cualquier estilo. El estilo que he ido mostrando en este libro es más bien imperativo, aunque con una tendencia clara hacia la orientación a objetos.

No es fácil definir qué es la programación orientada a objetos, pero he aquí algunas de sus características:

- Las definiciones de objetos (clases) a menudo se corresponden con objetos relevantes del mundo real. Por ejemplo, en el capítulo 12, la creación de la clase `BARAJA` fue un paso hacia la programación orientada a objetos.
- La mayoría de los métodos son métodos de objeto (aquellos que se aplican *sobre* un objeto, usando la notación punto) en lugar de métodos tradicionales (aquellos que reciben el objeto como un parámetro). Hasta ahora todos los métodos que hemos escrito han sido métodos que recibían los objetos como parámetros. En este capítulo escribiremos algunos métodos de objeto.
- La característica del lenguaje más asociada con la programación orientada a objetos es la **herencia**. Comentaré la herencia después en este capítulo.

Recientemente la programación orientada a objetos ha llegado a ser muy popular, y hay gente<sup>1</sup> que afirma que es superior a otros estilos en varios aspectos. Espero que al exponerte a una variedad de estilos haya conseguido darte las herramientas que necesitas para entender y evaluar esas afirmaciones.

### 13.2. Métodos de objeto

Hasta ahora hemos visto dos tipos de métodos en Eiffel: aquellos que invocan mediante el operador punto, y aquellos que no. Por ejemplo, el método `rounded` se utiliza para redondear un número `DOUBLE`, y se aplica *sobre* el número escribiendo un punto entre el número y el nombre del método, como en `(4.27).rounded`. Este método, por tanto, es un **método de objeto**.

---

<sup>1</sup>Uno de los más conocidos es Bertrand Meyer, el creador del lenguaje Eiffel.



Los métodos que hemos estado escribiendo hasta ahora no se invocan de esta manera. Por ejemplo, el método `ordenar_baraja` no se aplica *sobre* una baraja mediante la sintaxis `baraja.ordenar_baraja`, sino que como parámetro la baraja a ordenar, y se invoca mediante `ordenar_baraja(baraja)`.

Otro ejemplo de este tipo de métodos es el método predefinido `print`, que recibe como parámetro el objeto que hay que imprimir, así que usamos `print(numero)` y no `numero.print`.

Hasta ahora, todos los métodos que hemos escrito son métodos que se invocan de esta última forma: pasando como parámetro el objeto con el que trabajan. Sin embargo, en Eiffel, por su estilo de programación, es mucho más natural escribir los métodos de tal forma que éstos actúen *sobre* el objeto, mediante el operador punto, en lugar de recibir el objeto como parámetro. Además, por razones que quedarán más claras pronto, este tipo de métodos a menudo son más cortos que sus correspondientes métodos escritos “al viejo estilo”.

### 13.3. El objeto actual

Cuando invocas a un método sobre un objeto, el objeto se convierte en el **objeto actual**. Dentro del método, puedes referirte a los atributos del objeto actual simplemente por sus nombres, sin tener que especificar el nombre del objeto.

Además, puedes hacer referencia al objeto actual usando la palabra clave `Current`. Esto puede ser útil en algunas ocasiones.

### 13.4. Números complejos

Para seguir el mismo ejemplo durante lo que queda de capítulo, consideraremos una definición de clase para números complejos. Los números complejos son útiles en muchas ramas de las matemáticas y de la ingeniería, y muchas operaciones se realizan usando aritmética de números complejos. Un número complejo es la suma de una parte real y una parte imaginaria, y a menudo se escribe de la forma  $x + yi$ , donde  $x$  es la parte real,  $y$  es la parte imaginaria, y  $i$  representa la raíz cuadrada de  $-1$ . Así,  $i \cdot i = -1$ .

La siguiente es una definición de clase para un nuevo tipo de objeto llamado `COMPLEJO`:

```
class COMPLEJO

  creation
    make,
    make_par

  feature
    real, imag: DOUBLE

  make is
    do
      real := 0
      imag := 0
    end

  make_par(re, im: DOUBLE) is
    do
      real := re
      imag := im
```

```

        end
    end
end

```

No debe haber nada sorprendente aquí. Los atributos son dos `DOUBLE`s que contienen las partes real e imaginaria. Los dos constructores son los normales: uno no recibe parámetros y asigna valores por defecto a los atributos, y el otro recibe parámetros idénticos a los atributos.

En el método `raíz`, o en cualquier parte donde queramos crear objetos `COMPLEJO`s, podemos crear el objeto y al mismo tiempo asignarles valores a sus atributos:

```

class PRINCIPAL    -- clase raíz

creation make

feature
    make is        -- método raíz
        local
            x: COMPLEJO
        do
            create x.make_par(3.0, 4.0)
        end
    end
end

```

### 13.5. Una función sobre números `COMPLEJO`s

Echemos un vistazo a las operaciones que podemos realizar sobre números complejos. El valor absoluto de un número complejo se define como  $\sqrt{x^2 + y^2}$ . El método `abs` es una función pura que calcula el valor absoluto. Escrita como un método externo, sería algo así:

```

abs(c: COMPLEJO): DOUBLE is
do
    Result := (c.real * c.real + c.imag * c.imag).sqrt
end

```

Esta versión de `abs` calcula el valor absoluto de `c`, el objeto `COMPLEJO` que recibe como parámetro. La siguiente versión de `abs` es un método de objeto; calcula el valor absoluto del objeto actual (el objeto sobre el que se va a invocar el método). Por tanto, no tiene que recibir ningún parámetro. Además, tenemos que incluir la definición de este método dentro de la definición de la clase `COMPLEJO`.

```

class COMPLEJO
    -- ...
feature
    -- ...
    abs: DOUBLE is
        do
            Result := (real * real + imag * imag).sqrt
        end
    end
end

```

He obviado el resto de la definición de la clase `COMPLEJO`, indicándolo mediante los puntos suspensivos. Observa que he eliminado el parámetro de `abs`, el cual ya no es necesario. Dentro del

método, puedo referirme a los atributos `real` e `imag` por su nombre sin tener que especificar el objeto. Eiffel sabe implícitamente que me estoy refiriendo a los atributos del objeto actual. Si quisiera dejarlo más explícito, podría haber usado la palabra clave `Current`:

```
abs: DOUBLE is
do
    Result := (Current.real * Current.real + Current.imag * Current.imag).sqrt
end
```

Pero queda más largo y en realidad menos claro. Para invocar a este método, lo invocamos sobre un objeto, por ejemplo:

```
local
    y: COMPLEJO
    res: DOUBLE
do
    create y.make_par(3.0, 4.0)
    res := y.abs
end
```

## 13.6. Otra función sobre números COMPLEJOS

Otra operación que podrías querer realizar sobre números complejos es la suma. Puedes sumar números complejos sumando parte real con parte real, y parte imaginaria con parte imaginaria. Escrito como un método externo, quedaría así:

```
suma(a, b: COMPLEJO): COMPLEJO is
do
    Result := create {COMPLEJO}.make_par(a.real + b.real, a.imag + b.imag)
end
```

Para invocar a este método, podemos pasar los dos operandos como argumentos:

```
local
    sum: COMPLEJO
do
    sum := suma(x, y)    -- 'x' e 'y' son objetos COMPLEJOS
end
```

Escrito como un método de objeto, sólo necesitaría recibir un argumento, que se sumará con el objeto por defecto:

```
class COMPLEJO
-- ...
feature
-- ...
    suma(b: COMPLEJO): COMPLEJO is
do
    Result := create {COMPLEJO}.make_par(real + b.real, imag + b.imag)
end
end
```

Otra vez, podemos referirnos implícitamente a los atributos del objeto actual, pero para referirnos a los atributos de **b** tenemos que nombrar a **b** explícitamente usando la notación punto. Para llamar a este método, lo invocamos sobre uno de los operandos y pasamos el otro como argumento:

```
local
  sum: COMPLEJO
do
  sum := x.suma(y)  -- 'x' e 'y' son objetos COMPLEJOS
end
```

A partir de estos ejemplos puedes ver que el objeto actual (**Current**) puede hacer el papel de uno de los parámetros. Por esta razón, el objeto actual a veces se denomina el parámetro **implícito**.

### 13.7. Un nombre más adecuado

Estarás de acuerdo conmigo en que `x.suma(y)` no es una notación muy adecuada para representar la suma de dos objetos. Lo más indicado sería usar el operador `+`, como ocurre con los enteros, los reales, etcétera, de forma que pudiéramos usar `x + y`. En Eiffel es posible hacer que el nombre de un método sea un operador. En este caso, sería el operador `+`. Este operador es *infijo*, lo que significa que se sitúa entre los dos operandos, como en `4 + 7`. También existen operadores *prefijo*, como el signo menos en `-9`, y operadores *postfijos*, como el factorial en `3!`.

El mismo método anterior, pero usando el operador infijo `+` como nombre, sería así:

```
class COMPLEJO
  -- ...
feature
  -- ...
  infix "+"(b: COMPLEJO): COMPLEJO is
    do
      Result := create {COMPLEJO}.make_par(real + b.real, imag + b.imag)
    end
end
```

De esta forma, para representar la suma de dos números complejos se utilizaría la notación tradicional:

```
local
  sum: COMPLEJO
do
  sum := x + y  -- 'x' e 'y' son objetos COMPLEJOS
end
```

Sencillo, ¿verdad? En todo caso, no olvides que la sintaxis `x + y` anterior significa que se aplica el método `"+"` sobre el objeto `x`, pasándole como parámetro el objeto `y`. Más o menos como si escribiéramos `x.infix"+"(y)`. Sólo cambia la sintaxis.

### 13.8. Un modificador

Como otro ejemplo más, veremos **conjugado**, que es un método modificador que transforma un número **COMPLEJO** en su complejo conjugado. El complejo conjugado de  $x + yi$  es  $x - yi$ .

Si lo escribimos como un método externo, quedaría así:

```

conjugado(c: COMPLEJO) is
do
    c.set_imag(-c.imag)
end

```

Para lo cual tenemos que suponer que existe un método llamado `set_imag` definido en la clase `COMPLEJO`, el cual asigna al atributo `imag` el valor que recibe como parámetro (en Eiffel no se puede hacer `c.imag := -c.imag`, porque la notación punto no puede aparecer en la parte izquierda de una asignación).

Si lo escribimos como un método de objeto, quedaría así

```

class COMPLEJO
-- ...
feature
-- ...
    conjugado is
    do
        imag := -imag
    end
end

```

Ya te habrás dado cuenta que convertir un método de un tipo en otro es un proceso mecánico. Con un poco de práctica, serás capaz de hacerlo sin pensar demasiado, lo que es bueno porque así no te sentirás restringido a escribir un método u otro. Deberías estar igual de familiarizado con ambos de forma que puedas siempre escoger el que parezca más apropiado para la operación que estás escribiendo.

Por ejemplo, las operaciones simples que se aplican sobre un objeto pueden escribirse de forma más concisa como métodos de objeto (incluso si reciben algunos argumentos adicionales).

### 13.9. El método `to_string`

Existen dos métodos de objeto que son comunes en muchos tipos de objeto: `to_string` y `is_equal`. `to_string` convierte el objeto en una representación razonable en forma de cadena para poder ser impresa. `is_equal` se utiliza para comparar objetos.

El método `to_string` está definido en todos los tipos básicos, como `INTEGER` o `DOUBLE`, así que, por ejemplo, `4.to_string` devolvería la cadena "4". Esto es algo que vamos a utilizar para definir el método `to_string` en la clase `COMPLEJO`.

Este podría ser el método `to_string` para la clase `COMPLEJO`:

```

class COMPLEJO
-- ...
feature
-- ...
    to_string: STRING is
    do
        Result := real.to_string + " + " + imag.to_string + "i"
    end
end

```

No confundas el método `to_string` aplicado sobre `real` o `imag` (que son objetos `DOUBLE`), con el método `to_string` que estamos definiendo en la clase `COMPLEJO`. Dos métodos definidos en clases distintas pueden tener el mismo nombre, y de hecho, suele hacerse para facilitar la lectura de los programas.

El tipo de retorno de `to_string` es `STRING`, naturalmente, y no recibe parámetros. Puedes invocar a `to_string` de la forma usual:

```
local
  x: COMPLEJO
  s: STRING
do
  create x.make_par(1.0, 2.0)
  s := x.to_string
  print(s)
end
```

En este caso, la salida es `1.0 + 2.0i`.

Esta versión de `to_string` no produce un buen resultado si la parte imaginaria es negativa. Como ejercicio, arrégalo.

### 13.9.1. El método `print_on`

Al terminar de leer el apartado anterior te puedes haber preguntado por qué no podemos simplemente pasar el objeto `COMPLEJO` como argumento al método `print`, para que sea éste el que lo imprima según hemos definido en el método `to_string`. Es decir: antes de imprimir el objeto, tenemos que crear una cadena con `to_string`, y luego imprimir esa cadena con `print`. ¿No podría hacerse directamente?

El problema es que el método `print` no sabe cómo imprimir ese nuevo tipo de objeto que estamos creando: los `COMPLEJO`s. Sin embargo, Eiffel tiene una solución para que podamos imprimir directamente los objetos sin necesidad de pasarlos previamente a cadenas. Lo malo es que dicha solución es un tanto complicada, sobre todo si, como te ocurre ahora, todavía no se ha visto nada sobre la *herencia*. Así que por ahora sólo voy a mostrarte cómo quedaría la definición de la clase `COMPLEJO` para que `print` admita e imprima correctamente un objeto de ese tipo. Resultaría más o menos así:

```
class COMPLEJO

inherit
  ANY
  redefine
    print_on
  end

  -- ...
feature
  -- ...
  to_string: STRING is
  do
    Result := real.to_string + " + i" + imag.to_string + "%N"
  end

  print_on(file: OUTPUT_STREAM) is
```

```

        do
            file.put_string(to_string)
        end
    end
end

```

Observa los cambios: en primer lugar, hemos incluido una nueva cláusula en la clase, llamada **inherit**. Ya veremos para qué sirve cuando estudiemos la herencia. Y también hemos añadido un nuevo método, llamado **print\_on**. El método **print**, cuando recibe como parámetro un objeto, lo que hace es invocar al método **print\_on** de ese objeto, para que sea el propio objeto el que se imprima a sí mismo.

En nuestro caso, la definición del método **print\_on** consiste simplemente en una llamada al método **put\_string** sobre el objeto **file**, que es un objeto que se recibe como parámetro y que representa la pantalla. El método **put\_string** también recibe un parámetro, que es la cadena que hay que imprimir, y que en nuestro caso será simplemente la salida del método **to\_string**. Gracias a esto, el método **print** ahora sabe cómo imprimir correctamente un objeto **COMPLEJO**, así que el siguiente código funcionaría correctamente:

```

local
    x: COMPLEJO
do
    create x.make_par(1.0, 2.0)
    print(x)
end

```

Imprimiendo  $1.0 + 2.0i$ , como antes.

## 13.10. El método **is\_equal**

Cuando usas el operador **=** para comparar dos objetos, en realidad lo que estás preguntando es “¿Son estas dos cosas el mismo objeto?”. Es decir si ambos objetos se refieren a la misma localización en memoria.

Para muchos tipos, esta no es la definición apropiada de igualdad. Por ejemplo, dos números complejos son iguales si son iguales sus partes reales y son iguales sus partes imaginarias.

Cuando creas un nuevo tipo de objeto, puedes proporcionar tu propia definición de igualdad creando un método de objeto llamado **is\_equal**. Para la clase **COMPLEJO**, éste queda así:

```

class COMPLEJO
    -- ...
feature
    -- ...
    is_equal(b: COMPLEJO): BOOLEAN is
    do
        Result := real = b.real and imag = b.imag
    end
end

```

Por convenio, **is\_equal** siempre es un método de objeto. El tipo de retorno debe ser **BOOLEAN**.

La documentación de **is\_equal** en la clase **ANY** proporciona algunas directrices que deberías tener en mente cuando hagas tu propia definición de igualdad:

El método **is\_equal** implementa una relación de equivalencia:

- Es reflexivo: para todo valor referencia `x`, `x.is_equal(x)` debe devolver `True`.
- Es simétrico: para cualesquiera valores referencia `x` e `y`, `x.is_equal(y)` debería devolver `True` si y sólo si `y.is_equal(x)` devuelve `True`.
- Es transitivo: para cualesquiera valores referencia `x`, `y`, y `z`, si `x.is_equal(y)` devuelve `True` y `y.is_equal(z)` devuelve `True`, entonces `x.is_equal(z)` debería devolver `True`.
- Es consistente: para cualesquiera valores referencia `x` e `y`, varias llamadas `x.is_equal(y)` devolverán consistentemente `True` o devolverán consistentemente `False`.
- Para cualquier valor referencia `x`, `x.is_equal(Void)` debería devolver `False`.

La definición de `is_equal` que he proporcionado satisface todas estas condiciones excepto una. ¿Cuál es? Como ejercicio, arrégalo.

### 13.11. Invocar a un método de objeto desde otro

Como podrías esperar, se permite y es común invocar a un método de objeto desde otro. Por ejemplo, para normalizar un número complejo, divides ambas partes por su valor absoluto. Puede que no sea obvio el por qué esto es útil, pero lo es.

Escribamos el método `normalizar` como un método de objeto, y hagamos que sea un modificador.

```
class COMPLEJO
  -- ...
feature
  -- ...
  normalizar is
    local
      d: DOUBLE
    do
      d := Current.abs
      real := real / d
      imag := imag / d
    end
end
```

La primera línea del método encuentra el valor absoluto del objeto actual llamando a `abs` sobre el objeto actual. En este caso indiqué el objeto actual explícitamente, pero podría no haberlo puesto. Si invocas a un método de objeto dentro de otro, Eiffel asume que estás invocándolo sobre el objeto actual.

Como ejercicio, reescribe `normalizar` como una función pura. Luego escríbelo como un método externo, fuera de la definición de la clase `COMPLEJO`.

### 13.12. Herencia

La característica del lenguaje que se asocia más a menudo con la programación orientada a objetos es la **herencia**. La herencia es la capacidad de definir una nueva clase que es una versión modificada de una clase previamente definida (incluidas las clases predefinidas “de fábrica”).



La principal ventaja de esta característica es que puedes añadir nuevos métodos o atributos a una clase existente sin tener que modificarla. Esto es particularmente útil para clases predefinidas, ya que no puedes modificarlas aunque quieras.

El motivo por el que la herencia se llama “herencia” es que la nueva clase hereda todos los atributos y métodos de la clase existente. Extendiendo esta metáfora, la clase existente a menudo se llama la clase **precursora**.

### 13.13. Rectángulos dibujables

Como ejemplo de herencia, vamos a coger la clase **RECTANGULO** existente y vamos a hacer que sea “dibujable”. Esto es, vamos a crear una nueva clase llamada **RECTANGULO\_DIBUJABLE** que tendrá todos los atributos y métodos de un **RECTANGULO**, más un método adicional llamado **dibujar** que dibujará el rectángulo por la pantalla.

La definición de clase se parece a ésto:

```
class RECTANGULO_DIBUJABLE

inherit
    RECTANGULO

feature
    dibujar is
        local
            i, j: INTEGER
        do
            from
                i := 0
            until
                i >= ancho - x
            loop
                from
                    j := 0
                until
                    j >= alto - y
                loop
                    print("#")
                    j := j + 1
                end
                print("%N")
                i := i + 1
            end
        end
    end

end
```

Sí, esto es realmente todo lo que hay en la definición de clase. La segunda línea indica que **RECTANGULO\_DIBUJABLE** hereda de **RECTANGULO**. La palabra clave **inherit** se utiliza para identificar la clase precursora.

El resto es la definición del método **dibujar**, el cual hace referencia a los atributos **x**, **y**, **ancho** y **alto**. Puede parecer extraño referirse a unos atributos que no aparecen en la definición de la clase, pero recuerda que se están heredando de la clase precursora.

Para crear y dibujar un **RECTANGULO\_DIBUJABLE**, podrías usar lo siguiente:

```

class RAIZ

  creation make

  feature
    make is
      local
        rd: RECTANGULO_DIBUJABLE
      do
        create rd.make(10, 10, 200, 200)
        rd.dibujar
      end
    end
  end
end

```

Puede parecer extraño usar la sentencia `create` y el método `make` sobre un objeto que no tiene constructores. `RECTANGULO_DIBUJABLE` hereda los constructores de su clase precursora, así que no hay ningún problema.

Cuando llamemos a `dibujar`, Eiffel llamará al método definido en `RECTANGULO_DIBUJABLE`. Si llamamos a `mover` o a algún otro método de `RECTANGULO` sobre `dr`, Eiffel debe saber usar el método definido en la clase precursora.

### 13.14. La jerarquía de clases

En Eiffel, todas las clases heredan de alguna otra clase. La clase más básica se llama `ANY`, y proporciona métodos como `is_equal`, `to_string` o `print`, entre otros.

Muchas clases heredan de `ANY`, incluidas casi todas las clases que hemos escrito y muchas de las clases predefinidas. Toda clase que no hereda explícitamente de ninguna clase, en realidad es heredera por defecto de `ANY`. Por eso precisamente podemos, entre otras cosas, usar el método `print` en cualquier método de cualquier clase.

Algunas ramas hereditarias son largas, no obstante. Por ejemplo, `TWO_WAY_LINKED_LIST` hereda de `LINKED_COLLECTION`, que hereda de `COLLECTION`, que hereda de `SAFE_EQUAL`, que hereda de `ANY`. No importa lo larga que sea la cadena: `ANY` es siempre la clase precursora final de todas las clases.

Todas las clases en Eiffel pueden ser organizadas en un “árbol familiar” que se llama la jerarquía de clases. `ANY` suele aparecer en la parte superior, con todas las clases “hijas” debajo.

Además de la clase `ANY`, en Eiffel también se tiene la clase `NONE`. Tiene la particularidad de que todas las clases que no tienen clases herederas son precursoras de `NONE`. Es decir, `NONE` heredan, directa o indirectamente, de todas las clases que aparecen en nuestro programa. Por tanto, en la jerarquía de clases, `NONE` siempre aparece en la parte inferior, como “hija” de todas las clases que no tengan descendentes.

En definitiva, y para ser precisos, la rama hereditaria que comenté antes debería empezar por `NONE`, que hereda de `TWO_WAY_LINKED_LIST`, que hereda de `LINKED_COLLECTION`, que hereda de `COLLECTION`, que hereda de `SAFE_EQUAL`, que hereda de `ANY`. Es decir: todas las ramas de la jerarquía de clases empiezan siempre por `NONE`, y van subiendo hasta acabar en `ANY`.

### 13.15. Diseño orientado a objetos

La herencia es una característica poderosa. Algunos programas que serían muy complicados sin la herencia pueden ser escritos de manera concisa y simple con ella. Además, la herencia

puede facilitar la reutilización del código, ya que puedes adaptar el comportamiento de las clases predefinidas sin tener que modificarlas.

Por otra parte, la herencia puede hacer que los programas sean difíciles de leer, ya que a veces no está claro, cuando se invoca a un método, dónde encontrar la definición. Por ejemplo, uno de los métodos que puedes invocar sobre un objeto de tipo `TWO_WAY_LINKED_LIST` es `replace_all`. ¿Cómo sabes si ese método está definido en la clase `TWO_WAY_LINKED_LIST` o si por el contrario está definido en alguna de sus muchas clases precursoras? La única forma es echar un vistazo a la documentación.

La mejor documentación para una clase es la **forma corta** o **forma plana** de la clase. La forma plana de una clase incluye todas las características (atributos y métodos) propias de la clase, junto con las características heredadas de todas las demás clases precursoras, agrupadas por el nombre de la clase en la que fueron definidas esas características. De esta forma, siempre se puede comprobar con facilidad en qué clase se encuentra la definición de una característica.

De todas formas, muchas de las cosas que pueden hacerse usando la herencia también pueden hacerse de una manera casi tan elegante (o más incluso) sin ella.

### 13.16. Glosario

**método de objeto:** Un método que se invoca sobre un objeto, y que opera sobre ese objeto, al que se refiere por la palabra clave `Current` en Eiffel o “el objeto actual” en castellano. Los métodos de objeto se definen dentro de la clase del objeto sobre el que actúan, y se invocan usando la notación punto.

**método externo:** Un método que se define fuera de la clase del objeto sobre el que actúa. Estos métodos no se invocan sobre objetos.

**objeto actual:** El objeto sobre el que se invoca un método de objeto. Dentro de ese método, el objeto actual se identifica por `Current`.

**Current:** La palabra clave que hace referencia al objeto actual.

**implícito:** Todo lo que no se dice o queda implicado. Dentro de un método de objeto, puedes hacer referencia a los atributos de forma implícita (sin tener que indicar el nombre del objeto).

**explícito:** Todo lo que se tiene que decir completamente. Dentro de un método externo, todas las referencias a los atributos tienen que ser explícitas.

## Capítulo 14

# Listas enlazadas

### 14.1. Referencias en objetos

En el último capítulo vimos que los atributos de un objeto pueden ser arrays, y mencioné que pueden ser otros objetos, también.

Una de las posibilidades más interesantes es la de que un objeto puede contener una referencia a otro objeto del mismo tipo. Existe una estructura de datos clásica, la **lista**, que aprovecha esta característica.

Las listas están hechas de **nodos**, donde cada nodo contiene una referencia al siguiente nodo de la lista. Además, cada nodo a menudo contiene una unidad de información llamada la **carga**. En nuestro primer ejemplo, la **carga** va a ser un simple entero, pero más tarde escribiremos una lista **genérica** que puede contener objetos de cualquier tipo.

### 14.2. La clase NODO

Como es habitual cuando escribimos una nueva clase, comenzaremos con los atributos, uno o dos constructores y el método **to\_string** de forma que podamos probar el mecanismo básico de creación y visualización del nuevo tipo.

```
class NODO

  creation
    make,
    make_par

  feature
    carga: INTEGER
    sgte: NODO

    set_carga(c: INTEGER)
      do
        carga := c
      end

    set_sgte(s: NODO)
      do
```

```

        sgte := s
    end

    make is
    do
        carga := 0
        sgte := Void
    end

    make_par(c: INTEGER; s: NODO) is
    do
        carga := c
        sgte := s
    end

    to_string: STRING is
    do
        Result := carga.to_string
    end
end

```

Las declaraciones de los atributos se deducen naturalmente de la especificación, y el resto se deduce mecánicamente de los atributos. La expresión `carga.to_string`, por supuesto, convierte el número entero en una cadena.

Para probar la implementación que hemos hecho hasta ahora, podemos poner algo así en el método raíz:

```

local
    nod: NODO
do
    create nod.make_par(1, Void)
    print(nod.to_string)
end

```

El resultado es simplemente

1

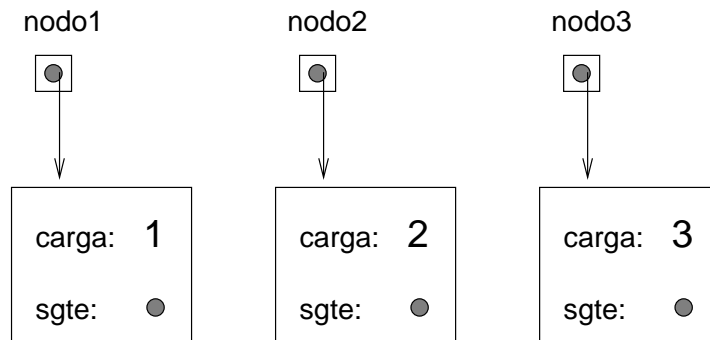
Para hacer la cosa más interesante, ¡necesitamos una lista con más de un nodo!

```

local
    nodo1, nodo2, nodo3: NODO
do
    create nodo1.make_par(1, Void)
    create nodo2.make_par(2, Void)
    create nodo3.make_par(3, Void)
end

```

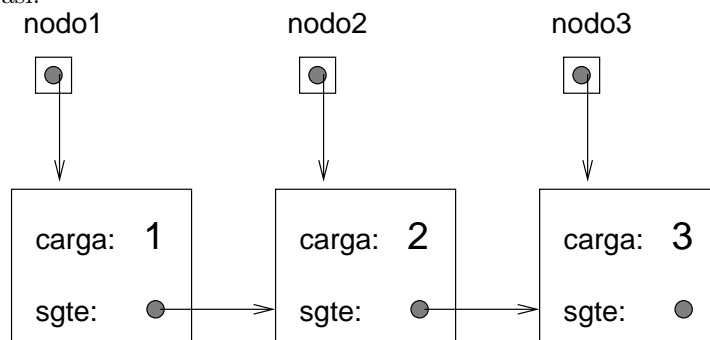
Este código crea tres nodos, pero todavía no tenemos una lista porque los nodos no están **enlazados**. El diagrama de estado tiene este aspecto:



Para enlazar los nodos, tenemos que hacer que el primer nodo haga referencia al segundo, y que el segundo haga referencia al tercero.

```
nodo1.set_sgte(nodo2)
nodo2.set_sgte(nodo3)
nodo3.set_sgte(Void)
```

La referencia del tercer nodo es `Void`, lo que indica que es el final de la lista. Ahora el diagrama de estado queda así:



Ahora sabemos cómo crear nodos y enlazarlos en listas. Lo que puede resultar poco claro en este momento es para qué.

### 14.3. Listas como colecciones

Lo que hace útiles a las listas que es son una manera de ensamblar varios objetos en una única entidad, a veces llamada colección. En el ejemplo, el primero nodo de la lista sirve como referencia a la lista completa.

Si queremos pasar la lista como parámetro, todo lo que tenemos que hacer es pasar una referencia al primer nodo. Por ejemplo, el método `imprimir_lista` recibe un único nodo como argumento. Comenzando por la cabeza de la lista, imprime cada nodo hasta que llega al final de la lista (indicada por la referencia a `Void`).

```
imprimir_lista(lista: NODO) is
  local
    nod: NODO
  do
    from
      nod := lista
    until
      nod = Void
```

```

loop
    print(nod.to_string)
    nod := nod.sgte
end
print("%N")
end

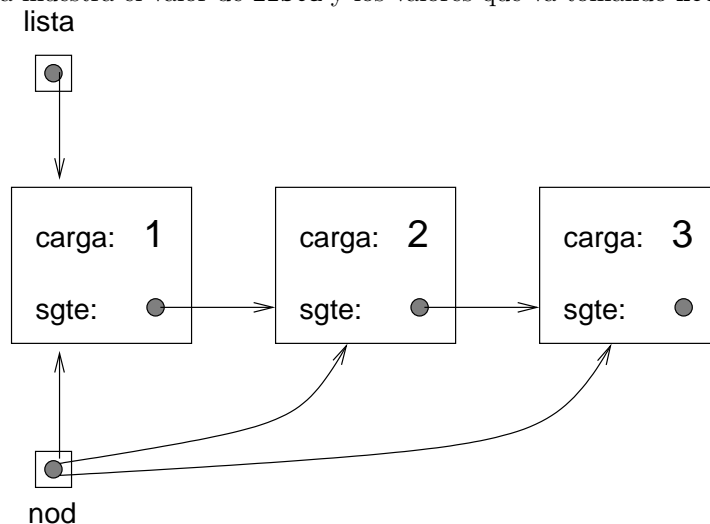
```

Para invocar a este método sólo tenemos que pasarle una referencia al primer nodo:

```
imprimir_lista(nodo1)
```

Dentro de `imprimir_lista` tenemos una referencia al primer nodo de la lista, pero ninguna variable se refiere a los demás nodos. Tenemos que usar el atributo `sgte` de cada nodo para obtener el nodo siguiente.

Este diagrama muestra el valor de `lista` y los valores que va tomando `nod`:



Esta manera de moverse a través de una lista se llama **recorrido**, exactamente igual que el patrón que seguimos para movernos a través de los elementos del array. Es típico usar una variable como `nod` que haga referencia a cada uno de los nodos de la lista en sucesión.

La salida de este método es

```
123
```

Por convenio, las listas se imprimen entre corchetes, con comas entre los elementos, como en `[1, 2, 3]`. Como ejercicio, modifica `imprimir_lista` de forma que genere la salida en ese formato.

## 14.4. Listas y recursividad

La recursividad y las listas van juntas como los langostinos y la manzanilla. Por ejemplo, este es un algoritmo recursivo para imprimir una lista al revés:

1. Divide la lista en dos partes: el primer nodo (denominado cabeza) y el resto (llamado cola).
2. Imprime la cola al revés.
3. Imprime la cabeza.

Por supuesto, el paso 2, la llamada recursiva, supone que tenemos una manera de imprimir una lista al revés. Pero *si* suponemos que la llamada recursiva funciona —el salto de fe— entonces podemos convencernos a nosotros mismos de que este algoritmo funciona.

Todo lo que necesitamos es un caso base, y una forma de probar que cualquier lista al final alcanzará el caso base. Una elección natural del caso base es una lista con un único elemento, pero una elección aún mejor es la lista vacía, representada por `Void`.

```
imprimir_al_reves(lista: NODO) is
  local
    cabeza, cola: NODO
  do
    if list /= Void then
      cabeza := lista
      cola := lista.sgte
      imprimir_al_reves(cola)
      print(cabeza)
    end
  end
end
```

La condición de la sentencia `if` maneja el caso base. Las dos primeras líneas dentro del `if` dividen la lista en `cabeza` y `cola`. Las últimas dos líneas imprimen la lista.

Invocamos a este método exactamente igual que cuando invocamos a `imprimir_lista`:

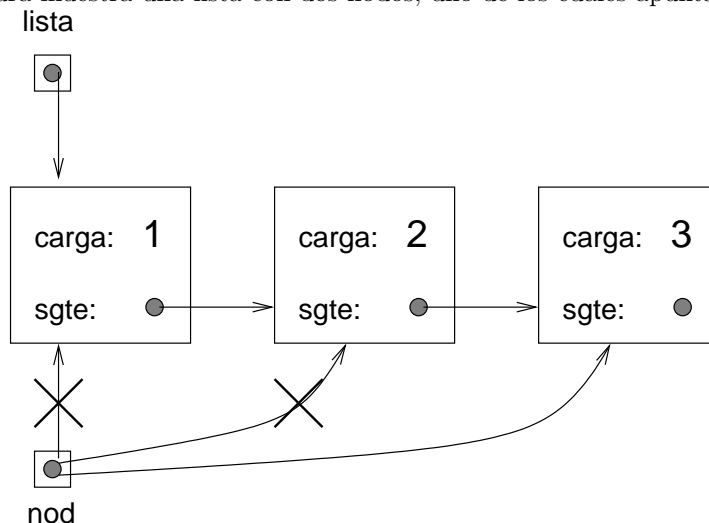
```
imprimir_al_reves(nodo1)
```

El resultado es la lista al revés.

¿Podemos asegurar que este método terminará siempre? En otras palabras, ¿alcanzará siempre el caso base? De hecho, la respuesta es no. Hay algunas listas que hacen que este método fracase.

## 14.5. Listas infinitas

Nada previene el que un nodo apunte a otro nodo anterior en la lista, incluido a sí mismo. Por ejemplo, esta figura muestra una lista con dos nodos, uno de los cuales apunta a sí mismo.





Si llamamos a `imprimir_lista` sobre esta lista, entrará en un bucle sin fin. Si invocamos a `imprimir_al_reves` entrará en recursión infinita. Este tipo de comportamiento hace difícil el trabajar con listas infinitas.

No obstante, en ocasiones son útiles. Por ejemplo, podemos representar un número como una lista de dígitos y usar una lista infinita para representar decimales periódicos, como en 12.57777777...

En todo caso, resulta problemático el que no podamos demostrar que `imprimir_lista` e `imprimir_al_reves` finalizan. Lo mejor que podemos hacer es plantearnos hipótesis como “Si la lista no contiene bucles, entonces estos métodos terminarán”. Este tipo de afirmaciones se denominan **precondiciones**. Imponen una restricción sobre uno de los parámetros y describe el comportamiento del método si se satisface la restricción. Veremos más ejemplos pronto.

## 14.6. El teorema fundamental de la ambigüedad

Hay una parte de `imprimir_al_reves` que puede hacer fruncir el ceño:

```
cabeza := lista
cola := lista.sgte
```

Después de la primera asignación, `cabeza` y `lista` tienen el mismo tipo y el mismo valor. Entonces, ¿por qué creamos una nueva variable?

La razón es que las dos variables juegan papeles diferentes. Pensamos en `cabeza` como en una referencia a un sólo nodo, y pensamos en `lista` como en una referencia al primer nodo de la lista. Estos “papeles” no son parte del programa; están en la mente del programador.

La segunda asignación crea una nueva referencia al segundo nodo de la lista, pero en este caso pensamos en él como en una lista. Por tanto, incluso aunque `cabeza` y `cola` son del mismo tipo, juegan papeles diferentes.

Esta ambigüedad es útil, pero puede hacer que los programas sean más difíciles de leer. A menudo utilizo variables como `nodo` y `lista` para documentar cómo pretendo usar una variable, y a veces creo variables adicionales para eliminar la ambigüedad.

También podría haber escrito `imprimir_al_reves` sin `cabeza` ni `cola`, pero considero que queda más difícil de leer:

```
imprimir_al_reves(lista: NODO) is
do
  if lista /= Void then
    imprimir_al_reves(lista.sgte)
    print(lista.to_string)
  end
end
```

Mirando las dos llamadas a funciones, tenemos que recordar que `imprimir_al_reves` trata su argumento como una lista y que `print` trata su argumento como un único objeto.

Ten siempre en mente el **teorema fundamental de la ambigüedad**:

Una variable que haga referencia a un nodo puede ser tratar a ese nodo como un único objeto o como el primero de una lista de nodos.

## 14.7. Otros métodos para nodos

Puedes estar preguntándote por qué `imprimir_lista` e `imprimir_al_reves` son métodos externos, en lugar de métodos de objeto. Ya afirmé que todo lo que puede hacerse con métodos externos también puede hacerse con métodos de objeto, y viceversa; es sólo una cuestión de qué forma es la más clara.

En este caso hay una razón legítima para elegir los métodos externos. Está permitido enviar `Void` como argumento a un método externo, pero no está permitido invocar a un método de objeto sobre un objeto `Void`.

```
local
  nod: NODO
do
  nod := Void
  imprimir_lista(nod)    -- permitido
  nod.imprimir_lista    -- Error: Target is Void.
end
```

Esta limitación hace difícil escribir código manipulador de listas en un estilo limpio y orientado a objetos. En todo caso, un poco más tarde veremos una forma de arreglar esto.

## 14.8. Modificar listas

Evidentemente una manera de modificar una lista es cambiar la carga de uno de sus nodos, pero las operaciones más interesantes son las que añaden, eliminan o cambian el orden de los nodos.

Como ejemplo, escribiré un método que elimina el segundo nodo de una lista y devuelve una referencia al nodo eliminado.

```
eliminar_segundo(lista: NODO): NODO
local
  primero, segundo: NODO
do
  primero := lista
  segundo := lista.sgte

  -- hace que el primer nodo apunte al tercero
  primero.sgte := segundo.sgte

  -- separa el segundo nodo del resto de la lista
  segundo.sgte := Void

  Result := segundo
end
```

De nuevo, estoy usando variables temporales para hacer el código más legible. Así se usa este método:

```
local
  eliminado: NODO
-- ...
```

```

do
  -- ...
  imprimir_lista(nodo1)
  eliminado := eliminar_segundo(nodo1)
  imprimir_lista(eliminado)
  imprimir_lista(nodo1)
end

```

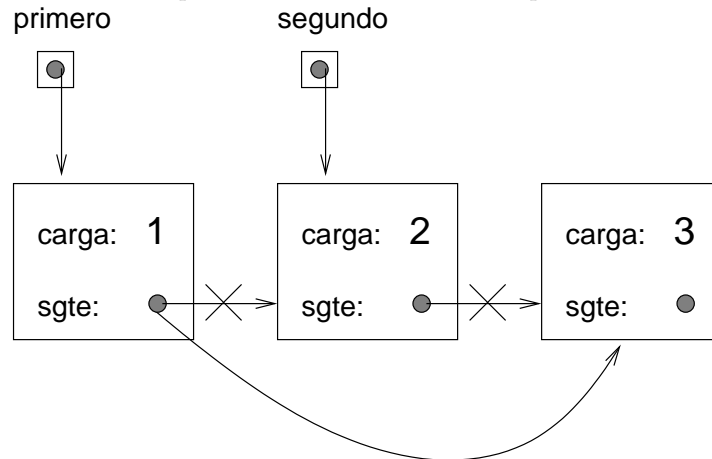
La salida es

```

[1, 2, 3]      la lista original
[2]            el nodo eliminado
[1, 3]         la lista modificada

```

Este es un diagrama de estado que muestra el efecto de esta operación.



¿Qué ocurre si llamamos a este método y le pasamos una lista con un único elemento? ¿Qué ocurre si pasamos la lista vacía como argumento? ¿Existe una precondition para este método?

## 14.9. Envoltorios y ayudantes

Para algunas operaciones de listas es útil dividir el trabajo en dos métodos. Por ejemplo, para imprimir una lista al revés con el formato convencional, [3, 2, 1] podemos usar el método `imprimir_al_reves` para imprimir 3, 2, pero necesitamos un método aparte para imprimir los corchetes y el primer nodo. Lo llamaremos `imprimir_al_reves_en_bonito`.

```

imprimir_al_reves_en_bonito(lista: NODO) is
  local
    cabeza, cola: NODO
  do
    print("[")

    if lista /= Void then
      cabeza := lista
      cola := lista.sgte
      imprimir_al_reves(colas)
      print(cabeza.to_string)
    end
  end

```

```

        print("] %N")
    end

```

De nuevo, es buena idea comprobar los métodos como éste para ver si funcionan con casos especiales como una lista vacía o con un único elemento.

En otra parte del programa, cuando usemos este método, llamaremos directamente a `imprimir_al_reves_en_bonito` y éste llamará a `imprimir_al_reves` por nosotros. En ese sentido, `imprimir_al_reves_en_bonito` actúa como un **envoltorio**, y éste usa a `imprimir_al_reves` como ayudante.

## 14.10. La clase LISTA\_ENLAZADA

Hay cierto número de sutiles problemas con la manera en la que hemos implementado las listas. Alternando causa y efecto, propondré una implementación alternativa primero y después explicaré qué problemas resuelve.

Primero, crearemos una nueva clase llamada `LISTA_ENLAZADA`. Sus atributos son un entero que contiene la longitud de la lista y una referencia al primer nodo de la lista. Los objetos `LISTA_ENLAZADA` sirven como asas o mangos para manipular listas de objetos `NODO`.

```

class LISTA_ENLAZADA

  creation make

  feature
    longitud: INTEGER
    cabeza: NODO

    make is
      do
        longitud := 0
        cabeza := Void
      end
    end

end

```

Una cosa agradable de la clase `LISTA_ENLAZADA` es que nos ofrece un espacio natural donde colocar funciones envoltorio como `imprimir_al_reves_en_bonito`, que puede convertirse en un método de objeto dentro de la clase `LISTA_ENLAZADA`.

```

class LISTA_ENLAZADA
  -- ...
  feature
    -- ...
    imprimir_lista_al_reves is
      local
        cola: NODO
      do
        print("[")

        if cabeza /= Void then
          cola := cabeza.sgte
          imprimir_al_reves(cola)
        end
      end
    end
  end
end

```

```

        print(cabeza)
    end
    print("] %N")
end
end

```

Para no confundir el método de `LISTA_ENLAZADA` con el de `NODO`, le he llamado `imprimir_lista_al_reves`. El método `imprimir_al_reves` sigue siendo el mismo que escribimos anteriormente para `NODO`. `imprimir_lista_al_reves` es el envoltorio, e `imprimir_al_reves` es el ayudante.

Así, uno de los beneficios de la clase `LISTA_ENLAZADA` es que proporciona un buen lugar para colocar las funciones envoltorio. Otro beneficio es que hace más fácil añadir o quitar el primer elemento de una lista. Por ejemplo, `insertar_primer` es un método de objeto para objetos `LISTA_ENLAZADA`; recibe un `INTEGER` como argumento y lo pone al principio de la lista.

```

class LISTA_ENLAZADA
  -- ...
  feature
    -- ...
    insertar_primer(i: INTEGER) is
      local
        nod: NODO
      do
        create nod.make_par(i, cabeza)
        cabeza := nod
        longitud := longitud + 1
      end
    end
end

```

Como siempre, para comprobar códigos como éste es una buena idea pensar en casos especiales. Por ejemplo, ¿qué ocurre si la lista está inicialmente vacía?

### 14.11. Invariantes

Algunas listas están “bien formadas”; otras no. Por ejemplo, si una lista contiene un bucle, provocará el fracaso de muchos de nuestros métodos, por lo que queremos obligar a que esas listas no contengan bucles. Otro requisito es que el valor de `longitud` en el objeto `LISTA_ENLAZADA` debería ser siempre igual al número real de objetos en la lista.

Los requisitos como éste se denominan **invariantes** porque, idealmente, deberían cumplirse para todos los objetos en todo momento. Especificar invariantes de objetos es una práctica útil en programación porque hace más fácil demostrar la corrección del código, comprobar la integridad de las estructuras de datos, y detectar errores.

Una cosa que en ocasiones confunde con las invariantes es que hay veces que no se cumplen. Por ejemplo, en la mitad de `insertar_primer`, después de haber añadido el nodo, pero antes de incrementar el valor de `longitud`, la invariante no se cumple. Este tipo de violación de la invariante es aceptable; de hecho, a menudo es imposible modificar un objeto sin modificar un invariante al menos durante un corto periodo de tiempo. Normalmente el requisito es que cada método que viole un invariante debe restaurar el invariante.

Si hay una considerable porción de código en la cual se incumple la invariante, es importante dejarlo claro en los comentarios, de manera que no se realice ninguna operación que dependa del invariante.

## 14.12. Invariantes en Eiffel

Una de las características del lenguaje Eiffel es que permite incluir los invariantes de cierto tipo de objeto dentro de la definición de clase para ese objeto. Los invariantes aparecen al final de la definición, dentro de la cláusula `invariant`. Por ejemplo, si queremos incluir el invariante de que la longitud de una lista tiene que ser una cantidad no negativa, lo haríamos de la siguiente forma:

```
class LISTA_ENLAZADA
  -- ...
feature
  -- ...
invariant
  longitud_no_negativa: longitud >= 0
end
```

Todos los invariantes se definen con una etiqueta (para distinguir unos de otros), seguido de dos puntos, y seguido por la condición que debe cumplirse. En este caso, la condición tiene la etiqueta `longitud_no_negativa`, y la condición es que el valor del atributo `longitud` ha de ser en todo momento mayor o igual que cero.

La condición del invariante debe ser una expresión lógica. Este invariante es bastante sencillo, pero hay algunos más complejos de escribir. Por ejemplo, el invariante que dice que el valor del atributo `longitud` debe coincidir en todo momento con el número de elementos de la lista.

Para escribir este invariante tenemos que desarrollar una función que cuente el número de elementos que tiene una lista. Supongamos que ya la tenemos definida, y que le ponemos el nombre de `cuenta_elementos`. En ese caso, la clase con sus invariantes podría escribirse así:

```
class LISTA_ENLAZADA
  -- ...
feature
  -- ...
  cuenta_elementos is
    do
      -- ...
    end

invariant
  longitud_no_negativa: longitud >= 0
  longitud_real: longitud = cuenta_elementos
end
```

¿Para qué puede servir incluir los invariantes dentro del código de la clase? Fundamentalmente, para aportar más información sobre el comportamiento y las propiedades de los objetos de esa clase; es decir: como parte de la documentación de esa clase.

Pero también pueden usarse como mecanismo que asegure que, durante la ejecución del programa, se cumplen los invariantes de las clases que forman el programa. Por tanto, podemos pedirle al programa que, durante su ejecución, compruebe constantemente sus invariantes, y que genere una excepción si alguno de ellos se incumple. Esto es muy útil mientras se está depurando el código.

Esto último también puede hacerse con las precondiciones y postcondiciones de los métodos, de tal manera que el programa compruebe en todo momento que se cumplen las precondiciones y postcondiciones correspondientes durante la llamada a un método.

### 14.13. Glosario

**lista:** Una estructura de datos que implementa una colección usando una secuencia de nodos enlazados.

**nodo:** Un elemento de una lista, normalmente implementado como un objeto que contiene una referencia a otro objeto del mismo tipo.

**carga:** La información contenida en un nodo.

**enlace:** Una referencia a objeto contenida dentro de otro objeto.

**estructura de datos genérica:** Un tipo de estructura de datos que puede contener datos de cualquier tipo.

**precondición:** Una condición que debe cumplirse para que el método funcione correctamente.

**invariante:** Una condición que debe cumplirse en un objeto en todo momento (excepto quizás mientras el objeto está siendo modificado).

**método envoltorio:** Un método que actúa como un intermediario entre un llamador y un método ayudante, a menudo ofreciendo un interface más limpio que el que ofrece el método ayudante.

# Capítulo 15

## Pilas

### 15.1. Tipos abstractos de datos

Los tipos de datos que hemos visto hasta ahora son todos concretos, en el sentido de que tenemos completamente especificados cómo están implementados. Por ejemplo, la clase **CARTA** representa una carta usando dos enteros. Como ya discutí en su momento, esta no es la única manera de representar una carta; existen muchas otras implementaciones alternativas.

Un **tipo abstracto de datos**, o TAD, especifica un conjunto de operaciones (o métodos) y la semántica de esas operaciones (lo que hacen), pero no especifican la implementación de esas operaciones. Ésto es lo que lo hace abstracto.

¿Por qué es útil eso?

- Simplifica la tarea de especificar un algoritmo si puedes denotar las operaciones que necesitas sin tener que pensar al mismo tiempo acerca de cómo se realizan las operaciones.
- Como suele haber muchas maneras de implementar un TAD, puede ser útil escribir un algoritmo que pueda ser utilizado con cualquiera de las posibles implementaciones.
- Los TADs bien conocidos, como el TAD Pila de este capítulo, a menudo se implementan en librerías estándar de forma que puedan escribirse una sola vez y ser usados por muchos programadores.
- Las operaciones sobre TADs proporcionan un lenguaje común de alto nivel para especificar y hablar acerca de algoritmos.

Cuando hablamos de TADs, a menudo distinguimos el código que utiliza el TAD, llamado **cliente**, del código que implementa el TAD, llamado **proveedor** porque proporciona un conjunto estándar de servicios. En Eiffel, cada TAD se implementa con una clase.

### 15.2. El TAD Pila

En este capítulo vamos a ver un TAD típico: la Pila. Una pila es una colección, lo que significa que es una estructura de datos que contiene varios elementos. Otras colecciones que hemos visto incluyen los arrays y las listas.

Como dije antes, un TAD se define por las operaciones que puedes realizar sobre él. Las pilas pueden realizar únicamente las siguientes operaciones:



**make:** Crea una nueva pila vacía.

**push:** Añade un nuevo elemento a la pila.

**cima:** Devuelve un elemento de la pila, sin modificar ésta. El elemento que se devuelve es siempre el último que se añadió.

**pop:** Quita un elemento de la pila. El elemento que se elimina es siempre el último que se añadió.

**es\_vacia:** Comprueba si la pila está vacía.

A las pilas a veces se las denomina estructuras de datos **LIFO**, o “Last In, First Out” (el primero en entrar es el último en salir), porque el último elemento añadido es el primero en ser sacado.

### 15.3. Genericidad

Observa que en la definición anterior del TAD Pila no indicamos en ningún momento el tipo de los elementos que contiene la pila. Esto es así porque la pila es un ejemplo de tipo de datos **genérico**. Un tipo de datos genérico es aquel que no especifica a priori alguno de los tipos sobre los que está definido. Esa ambigüedad nos permite definir las propiedades generales que toda pila posee, independientemente de los elementos que contenga.

Antes de usar un tipo genérico, tenemos que especificar todos los tipos que no aparezcan en su definición. En nuestro caso, para poder usar el TAD Pila tenemos antes que indicar el tipo que tendrán los elementos de la pila. En este sentido, un tipo genérico se parece a un método con parámetros: para poder usar el método, tenemos que pasarle valores a todos los parámetros del método. Igualmente ocurre con los tipos genéricos, sólo que sus parámetros son los tipos que no se especificaron en la definición del tipo genérico.

Normalmente, si el TAD representa una *colección*, el tipo que se deja sin especificar es precisamente el tipo de los elementos que forman la colección, con idea de que pueda admitir elementos de cualquier tipo. Evidentemente, los elementos de una colección son todos del mismo tipo. Nuestra pila, por ejemplo, en principio admite elementos de cualquier tipo, pero cuando la usemos podremos crear con ella pilas de enteros, pilas de cadenas, etcétera.

En Eiffel, las implementaciones de muchos TADs son genéricas, lo que significa que, si declaramos un objeto de ese tipo, tenemos que indicar el tipo de los elementos que contendrá. Esto se consigue usando corchetes. Vamos a ver un ejemplo, y de paso, echaremos un vistazo a una posible implementación del TAD Pila en Eiffel.

### 15.4. El objeto DS\_LINKED\_STACK

Eiffel proporciona, a través de la librería GOBO, un tipo de objeto predefinido llamado `DS_LINKED_STACK` que implementa el TAD Pila. Debes hacer un esfuerzo por mantener separadas ambas cosas —el TAD y la implementación Eiffel—. De hecho, como demostración de que el TAD y su implementación son cosas separadas, baste el hecho de que Eiffel dispone de dos implementaciones distintas del mismo TAD Pila: la clase `DS_LINKED_STACK`, que implementa el TAD por medio de listas enlazadas, y la clase `DS_ARRAYED_STACK`, que lo implementa mediante arrays. Ambas son implementaciones válidas del mismo TAD, por lo que podríamos perfectamente haber adoptado la segunda en favor de la primera.

Para nuestro primer ejemplo, crearemos y usaremos una pila de `NODOs`, según la definición de `NODO` vista en el capítulo anterior. La sintaxis para construir una nueva pila de `NODOs` es

```

local
  pila: DS_LINKED_STACK[NODO]
do
  create pila.make
end

```

Observa los corchetes a continuación del nombre de la clase en la declaración de `pila`. Los corchetes indican que `DS_LINKED_STACK` es una clase genérica. No basta con decir que `pila` es de tipo `DS_LINKED_STACK`: tenemos además que indicar el tipo de los elementos que contendrá la pila, y eso se indica entre corchetes junto al nombre de la clase. Si no ponemos nada entre corchetes, el programa no funcionará.

Observa ahora la palabra `NODO` entre corchetes tras el nombre de la clase en la declaración de `pila`. Esa declaración indica que los elementos de la pila deberán ser obligatoriamente de tipo `NODO`, y no de cualquier otro tipo.

Inicialmente la pila está vacía, como podemos confirmar con el método `is_empty`, que devuelve un `BOOLEAN`:

```

print(pila.is_empty)

```

Ahora podemos empezar a introducir elementos en la pila. Comenzaremos creando e imprimiendo una lista pequeña.

```

local
  pila: DS_LINKED_STACK[NODO]
  lista: LISTA_ENLAZADA
do
  create pila.make
  create lista.make
  lista.insertar_primer(3)
  lista.insertar_primer(2)
  lista.insertar_primer(1)
  lista.imprimir_lista
end

```

La salida es `[1, 2, 3]`. Para introducir un objeto `NODO` en la pila, usamos el método `put`:

```

pila.put(lista.cabeza)

```

El siguiente bucle recorre la lista y empuja todos los nodos en la pila:

```

from
  nod := lista.cabeza      -- nod es de tipo NODO
until
  nod = Void
loop
  pila.put(nod)
  nod := nod.sgte
end

```

Podemos consultar la pila con el método `item` para obtener el último elemento que se introdujo, y a continuación eliminar dicho elemento de la pila con el método `remove`.

```

elem := pila.item
pila.remove

```

¿Cuál debe ser el tipo de `elem`? Evidentemente, `elem` debe contener un valor del mismo tipo que los elementos de la pila. Como hemos declarado la pila de tal manera que sólo puede contener elementos de tipo `NODO`, tenemos que declarar la variable `elem` de ese mismo tipo `NODO`.

El siguiente bucle es un modismo típico usado para extraer todos los elementos de una pila, parando cuando queda vacía:

```

from
until
  pila.is_empty
loop
  print(pila.item.to_string) print(" ")
  pila.remove
end

```

La salida es 3 2 1. En otras palabras, ¡acabamos de usar una pila para imprimir al revés los elementos de una lista! De acuerdo, no es el formato estándar para imprimir una lista, pero usando una lista ha resultado ser considerablemente más fácil.

Deberías comparar este código con las implementaciones de `imprimir_lista_al_reves` del capítulo anterior. Existe un paralelismo natural entre la versión recursiva de `imprimir_lista_al_reves` y el algoritmo de pila de aquí. La diferencia es que `imprimir_lista_al_reves` utiliza la pila de ejecución para seguir la pista de los nodos mientras recorre la lista, y después los imprime a medida que retornan las llamadas recursivas. El algoritmo de pila hace lo mismo, sólo que usa un objeto `DS_LINKED_STACK` en lugar de la pila de ejecución.

## 15.5. Expresiones postfijas

En muchos lenguajes de programación, las expresiones matemáticas se escriben con el operador entre los dos operandos, como en `1+2`. Esta notación se llama **infija**. Una notación alternativa utilizada por algunas calculadoras se llama **postfija**. En notación postfija, el operador va después de los operandos, como en `1 2+`.

La razón por la cual a veces resulta útil la notación postfija es que existe una forma natural de evaluar una expresión postfija utilizando una pila.

- Comenzando al principio de la expresión, toma un término (operador u operando) cada vez.
  - Si el término es un operando, empújalo en la pila.
  - Si el término es un operador, saca dos operandos de la pila, realiza con ellos la operación correspondiente, y empuja el resultado en la pila.
- Cuando lleguemos al final de la expresión, sólo debería quedar en la pila un único operando. Ese operando es el resultado.

*Como ejercicio, aplica este algoritmo a la expresión `1 2 + 3 *`.*

Este ejemplo demuestra una de las ventajas de la notación postfija: no se necesitan usar paréntesis para controlar el orden de las operaciones. Para obtener el mismo resultado en notación infija, deberíamos haber escrito `(1 + 2) * 3`.

*Como ejercicio, escribe una expresión postfija que sea equivalente a `1 + 2 * 3`.*

## 15.6. Implementar TADs

Uno de los objetivos fundamentales de un TAD es separar los intereses del proveedor, quien escribe el código que implementa el TAD, y el cliente, quien usa el TAD. El proveedor sólo tiene que preocuparse de que la implementación sea correcta —de acuerdo con la especificación del TAD— y no de cómo va a ser usado.

Recíprocamente, el cliente *supone* que la implementación del TAD es correcta y no se preocupa de los detalles. Cuando utilizas una de las clases predefinidas de Eiffel, tienes el lujo de pensar exclusivamente como cliente.

Cuando implementas un TAD, por otra parte, también tienes que escribir código cliente para probarlo. En ese caso, a veces tienes que pensar cuidadosamente qué papel estás jugando en cada momento.

En los siguientes apartados, cambiaremos las tornas y veremos una manera de implementar el TAD Pila, usando un array. Comenzaremos pensando como un proveedor.

## 15.7. Más genericidad

La Pila es una estructura de datos genérica, por lo que sus elementos pueden ser, en principio, de cualquier tipo. Cuando se declara una variable de tipo Pila es cuando se especifica el tipo concreto que van a tener los elementos de esa pila. Por ejemplo:

```
local
  pila: DS_LINKED_STACK[DOUBLE]
```

`pila` sería una pila cuyos elementos son de tipo `DOUBLE`. Y si declaramos:

```
local
  otra_pila: DS_LINKED_STACK[STRING]
```

entonces `otra_pila` sería una pila en la que se almacenarían cadenas. Por tanto, el tipo de los elementos de la pila lo indicamos a continuación del nombre de la clase (`DS_LINKED_STACK`), encerrado entre corchetes. A ese tipo de datos, que hay que indicar cuando declaramos una pila, se le llama **parámetro actual genérico**.

Para que podamos usar parámetros actuales genéricos, la clase correspondiente tiene que estar diseñada para aceptarlos. Una clase que acepta parámetros actuales genéricos se denomina clase **genérica**.

Para definir una clase genérica tenemos que utilizar un nombre que represente a cualquier posible parámetro actual genérico, que en principio puede ser cualquiera, y que luego se concretará cuando se quiera crear un objeto de esa clase. Ese nombre se denomina **parámetro formal genérico**, y cumple la misma función que los parámetros en un método, sólo que en lugar de representar un posible valor, está representando un posible tipo.

El parámetro formal genérico de una clase aparece en la definición de la clase a continuación del nombre de la misma y encerrada entre corchetes. Como no representa a ningún tipo en concreto, sino a cualquiera de ellos, su nombre no debe coincidir con el de ningún tipo conocido.

Por ejemplo, si miramos la definición de la clase `DS_LINKED_STACK`, su primera línea podría ser la siguiente:

```
class DS_LINKED_STACK[G]
```

donde `G` es el parámetro formal genérico.

## 15.8. Implementación del TAD Pila basada en array

La implementación de nuestra pila basada en array debería basarse en una clase genérica, de manera que podamos luego utilizarla para poder crear pilas cuyos elementos sean de cualquier tipo, y que ese tipo se determine cuando se declaren variables que almacenen pilas.

Uno de los atributos de la implementación de la Pila es un array que usaremos para almacenar los elementos de la pila. Los elementos del array, por tanto, serán del mismo tipo que los elementos de la pila. Como aún no sabemos cuál es ese tipo concreto, lo representamos con el nombre del parámetro formal genérico. Si una clase tiene un único parámetro formal genérico, es costumbre que éste se llame **G**. Por tanto, el array contenido en la definición de la pila será un array de **G**.

La implementación de Pila también tendrá como atributo un índice entero que llevará la cuenta del último espacio utilizado en el array. Inicialmente, el array está vacío y el índice es 0.

Para añadir un elemento a la pila (**push**), incrementaremos el índice y copiaremos una referencia al elemento dentro de la pila. Para eliminar un elemento (**pop**) tan sólo tenemos que decrementar el índice. Y para devolver el último elemento introducido (**cima**), lo que haremos será devolver el elemento almacenado en el lugar indicado por el índice dentro del array.

Esta es la definición de la clase:

```
class PILA[G]

  creation make

  feature
    vector: ARRAY[G]
    indice: INTEGER

  make is
    do
      create vector.make(1, 128)
      indice := 0
    end
  end

end
```

Observa que tras el nombre de la clase aparece el parámetro formal genérico encerrado entre corchetes. Eso indica que la clase **PILA** es genérica, y que por tanto, cuando queramos crear una variable de tipo **PILA**, tendremos que indicar a continuación el nombre de un tipo de datos, entre corchetes. Por ejemplo, si queremos crear una pila de enteros, se hará así:

```
local
  mi_pila: PILA[INTEGER]
```

Aquí, **INTEGER** es el parámetro actual genérico, que sustituye al parámetro formal genérico **G** en la definición de la clase. Es decir: para el objeto **mi\_pila**, es como si la definición de la clase **PILA** tuviera escrita la palabra **INTEGER** en lugar de **G**. Así que **mi\_pila** tendrá un atributo llamado **vector** cuyo tipo es **ARRAY[INTEGER]**.

Si declaremos otra pila:

```
local
  mi_pila: PILA[INTEGER]
  tu_pila: PILA[DOUBLE]
```

Ahora tenemos dos pilas. Para el objeto `mi_pila`, el atributo `vector` es de tipo `ARRAY[INTEGER]`, pero para `tu_pila`, el mismo atributo es de tipo `ARRAY[DOUBLE]`. Esto es porque el parámetro formal genérico, `G`, se sustituye en cada caso por un parámetro actual genérico distinto.

Como es usual, una vez que hemos escogido los atributos, escribir un constructor es un proceso mecánico. Por ahora, el tamaño por defecto es 128 elementos. Más tarde consideraremos mejores formas de hacer esto.

Comprobar si la pila está vacía es trivial.

```
class PILA[G]
  -- ...
  feature
    -- ...
    es_vacia: BOOLEAN is
      do
        Result := indice = 0
      end
  end
end
```

Inicialmente, el tamaño de la pila es 128, pero el número de elementos es cero.

Las implementaciones de `push`, `pop` y `cima` surgen de forma natural a partir de la especificación.

```
class PILA[G]
  -- ...
  feature
    -- ...
    push(elem: G) is
      do
        indice := indice + 1
        vector.put(elem, indice)
      end

    pop is
      do
        indice := indice - 1
      end

    cima: G is
      do
        Result := vector @ indice
      end
  end
end
```

Observa cómo usamos el parámetro formal genérico `G` en las definiciones de los métodos `push` y `cima`. El argumento de `push` es el elemento que se introduce en la pila, y éste debe ser obligatoriamente del mismo tipo que los elementos del array `vector`; es decir: un tipo que todavía no sabemos cuál es, y que se especificará cuando se declare una variable de tipo `PILA`. Igualmente, cuando devolvemos el elemento situado en la cima de la pila, éste elemento ha de ser del mismo tipo que los elementos de `vector`; por eso `cima` devuelve un valor de tipo `G`.

La ventaja de usar un parámetro formal genérico es que la misma definición de clase sirve para crear pilas de cualquier tipo de elementos, y por tanto no hay que crear definiciones de clases separadas para pilas de enteros, pilas de reales, pilas de cadenas, pilas de `NODOS`, ...

## 15.9. Cambiar el tamaño de los arrays

Un inconveniente de esta implementación es que elige un tamaño arbitrario para el array cuando se crea la PILA. Si el usuario empuja más de 128 elementos en la pila, provocará un error de ejecución.

Una alternativa es dejar que el código cliente especifique el tamaño del array. Esto alivia el problema, pero requiere que el cliente sepa antes de tiempo cuántos elementos necesitará, y eso no siempre es posible.

Una solución mejor es comprobar si el array está lleno y hacerlo más grande cuando sea necesario. Ya que no tenemos idea de lo grande que hace falta que sea el array, una estrategia razonable es comenzar con un tamaño pequeño y doblarlo cada vez que se sobrepase.

Esta es la versión mejorada de `push`:

```
class PILA[G]
  -- ...
  feature
    -- ...
    push(elem: G) is
      do
        if lleno then
          cambia_tamano
        end
        -- aquí podemos demostrar indice < vector.upper
        indice := indice + 1
        vector.put(elem, indice)
      end
    end
end
```

Antes de insertar el nuevo elemento en el array, comprobamos si el array está lleno. De ser así, llamamos a `cambiar_tamano`. Después de la sentencia `if`, ya sabemos que: o bien (1) había sitio en el array, o (2) el array ha cambiado de tamaño y ahora hay sitio. Si `lleno` y `cambiar_tamano` son correctos, entonces podemos demostrar que `indice < vector.upper`, y por tanto la siguiente sentencia no provocará un error de ejecución.

Ahora todo lo que tenemos que hacer es implementar `lleno` y `cambiar_tamano`.

```
class PILA[G]
  -- ...
  feature
    -- ...
  feature {NONE}
    lleno: BOOLEAN is
      do
        Result := indice = vector.upper
      end

    cambiar_tamano is
      do
        vector.resize(1, vector.upper * 2)
      end
    end
end
```

El método **resize** está predefinido en cualquier objeto de tipo array, y lo que hace es reajustar el tamaño del mismo de forma que los elementos que no se ven afectados por el cambio de tamaño permanecen inalterados en el array. Como lo que estamos haciendo es duplicar el tamaño del array, es evidente que se mantienen todos sus anteriores elementos.

Observa que los métodos **lleno** y **cambiar\_tamano** están definidos en otra cláusula **feature**, distinta de aquella en la que están todos los demás métodos, y que tiene la curiosidad de que a continuación de la palabra **feature** aparece **{NONE}**. Eso significa que los métodos definidos a continuación no podrán ser invocados desde otra clase; sólo desde **PILA**. Esto es aceptable, ya que no hay ninguna razón para que el código cliente use estas funciones, y deseable, ya que refuerza la frontera entre la implementación y el cliente.

La implementación de **lleno** es trivial; simplemente comprueba si el índice se ha salido del rango de los índices válidos.

La implementación de **cambiar\_tamano** está muy clara, con el detalle de que asume que el array antiguo está lleno. En otras palabras, esa suposición es una precondition de este método. Es fácil ver que esta precondition se satisface, ya que la única manera de hacer que **cambiar\_tamano** se invoque es que **lleno** devuelva **True**, lo que sólo puede ocurrir si **indice = vector.upper**.

En el método **cambiar\_tamano**, ampliamos el array para hacerlo el doble de grande de como era antes. El nuevo **vector.upper** es el doble de grande de como era antes, e **indice** no ha cambiado, por lo que ahora debe cumplirse que **indice < vector.upper**. Esta expresión es una **postcondición** de **cambiar\_tamano**: algo que debe cumplirse cuando el método finalice (siempre y cuando se haya cumplido la precondition).

Precondiciones, postcondiciones e invariantes son herramientas útiles para analizar programas y demostrar que son correctos. En este ejemplo he mostrado un estilo de programación que facilita el análisis del programa y un estilo de documentación que ayuda a demostrar su corrección.

## 15.10. Glosario

**tipo abstracto de datos (TAD):** Un tipo de datos (normalmente una colección de objetos) que se define por un conjunto de operaciones, pero que puede implementarse de varias formas.

**cliente:** Un programa que usa un TAD (o la persona que escribió el programa).

**proveedor:** El código que implementa un TAD (o la persona que lo escribió).

**feature {NONE}:** Una cláusula de Eiffel que indica que los métodos y atributos definidos a continuación no pueden ser accedidos desde fuera de la clase actual.

**infixo:** Una forma de escribir expresiones matemáticas con los operadores entre los operandos.

**postfijo:** Una forma de escribir expresiones matemáticas con los operadores después de los operandos.

**parse:** To read a string of characters or tokens and analyze their grammatical structure.

**token:** A set of characters that are treated as a unit for purposes of parsing, like the words in a natural language.

**delimitador:** A character that is used to separate tokens, like the punctuation in a natural language.

**predicado:** Una expresión matemática que es verdadera o falsa.

**postcondición:** Un predicado que debe ser cierto al final de un método (siempre y cuando las precondiciones fueran ciertas al principio).



## Capítulo 16

# Colas

Este capítulo presenta dos TADs: Colas y Colas con prioridad. En la vida real una **cola** es una fila de consumidores esperando un servicio. En muchos casos, el primer consumidor de la fila es el siguiente que será servido. Hay excepciones, no obstante. Por ejemplo, en los aeropuertos, los consumidores cuyo vuelo despegará de manera inminente, a veces son seleccionados de la mitad de la cola. Además, en los supermercados un consumidor educado puede dejar que otro pase delante de él si tiene pocos artículos para comprar.

La regla que determina quién va a continuación se denomina **disciplina de encolado**. La disciplina de encolado más sencilla se llama **FIFO**, que significa “primero en entrar, primero en salir”. La disciplina de encolado más general es el **encolado por prioridad**, en el que cada consumidor tiene asignado una prioridad, y el que posee la prioridad más alta va primero, sin tener en cuenta el orden de llegada. La razón por la que he dicho que es la disciplina más general es que la prioridad puede basarse en cualquier cosa: a qué hora despegue el avión, cuántos artículos lleva el consumidor, o cómo de importante es éste. Por supuesto, no todas las disciplinas de encolado son “justas”, pero la justicia depende del cristal con el que se mire.

El TAD Cola y el TAD Cola con prioridad tienen el mismo conjunto de operaciones y sus interfaces son iguales. La diferencia está en la semántica de las operaciones: una Cola usa la política FIFO, y una Cola con prioridad (como su propio nombre sugiere) usa la política de encolado por prioridad.

Como ocurre con muchos TADs, hay varias maneras de implementar colas. Como una cola es una colección de elementos, podemos usar cualquiera de los mecanismos básicos para almacenar colecciones: arrays o listas. Nuestra elección estará basada en parte en sus prestaciones —cuánto se tarda en realizar las operaciones que queremos realizar— y en parte en la facilidad de implementación.

### 16.1. El TAD Cola

El TAD Cola se define por las siguientes operaciones:

**make:** Crea una nueva cola vacía.

**insertar:** Añade un nuevo elemento a la cola.

**cabeza:** Devuelve un elemento de la cola. El elemento que se devuelve es el primero que se insertó.

**eliminar:** Elimina un elemento de la cola. El elemento que se elimina es el primero que fue insertado.

**es\_vacia:** Comprueba si la cola está vacía.

Para mostrar una implementación de Cola, utilizaré la clase `LISTA_ENLAZADA` del capítulo 14. Además, supondré que tenemos una clase llamada `CONSUMIDOR` que define toda la información de cada consumidor, y las operaciones que podemos realizar sobre los consumidores.

Hasta donde llega nuestra implementación, no importa qué tipo de objetos contendrá la `COLA`, por lo que podemos hacer que ésta sea genérica. Este es el aspecto de la implementación:

```
class COLA[G]

  creation
    make

  feature
    lista: LINKED_LIST[G]

    make is
      do
        create lista.make
      end

    es_vacia: BOOLEAN is
      do
        Result := lista.is_empty
      end

    cabeza: G is
      do
        Result := item
      end

    insertar(elem: G) is
      do
        lista.add_last(elem)
      end

    eliminar is
      do
        lista.remove_first
      end
  end
end
```

Observa que hemos utilizado la clase predefinida `LINKED_LIST`, que implementa una lista enlazada genérica, de tal manera que sus elementos no son nodos, sino cualquier tipo de elemento indicado por su parámetro actual genérico, que en este caso se llama `G`, y coincide con el parámetro formal genérico de la clase `COLA`.

Un objeto cola contiene un único atributo, que es la lista que la implementa. Para los demás métodos, todo lo que tenemos que hacer es invocar uno de los métodos de la clase `LINKED_LIST`.

## 16.2. Envoltorio

Una implementación como esta se denomina **envoltorio**. Como en la vida real, un envoltorio es algo que oculta los detalles de una implementación y proporciona un interfaz más simple, o más

estándar.

Este ejemplo muestra una de las cosas buenas de los envoltorios, y es que es fácil de implementar, y uno de los riesgos de usar envoltorios, ¡que es el **riesgo en eficiencia!**

Normalmente, cuando llamamos a un método no estamos interesados en los detalles de su implementación. Pero hay un “detalle” que tal vez queramos conocer —la eficiencia del método—. ¿Cuánto tarda en realizar su trabajo, en función del número de elementos de la lista?

Veamos primero una implementación de `remove_first`.

```
class LINKED_LIST[G]
  -- ...
  feature
    -- ...
    remove_first is
      do
        if head /= Void then
          head := head.sgte
        end
      end
    end
end
```

Aquí no hay bucles ni llamadas a función, lo que sugiere que el tiempo de ejecución de este método es siempre el mismo. Un método así se dice que es una operación **constante en el tiempo**. En realidad, el método podría ser ligeramente más rápido cuando la lista está vacía, porque se salta el cuerpo de la sentencia alternativa, pero esa diferencia no es significativa.

El rendimiento de `add_last` es muy diferente.

```
class LINKED_LIST[G]
  -- ...
  feature
    -- ...
    add_last(obj: G) is
      local
        last: NODO
      do
        -- caso especial: lista vacía
        if head = Void then
          create head.make_par(obj, Void)
        else
          from
            last := head
          until
            last.sgte = Void
          loop
            -- recorre la lista para buscar el último nodo
            last := last.sgte
          end
          create last.sgte.make(obj, Void)
        end
      end
    end
end
```

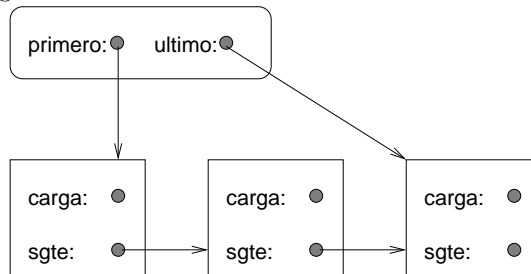
La primera rama de la sentencia alternativa gestiona el caso especial de añadir un nuevo nodo a una lista vacía. En este caso, de nuevo, el tiempo de ejecución no depende de la longitud de la lista.

En el caso general, por contra, tenemos que recorrer la lista para encontrar el último elemento de forma que podamos hacer que apunte al nuevo nodo.

Este recorrido conlleva un tiempo proporcional a la longitud de la lista. Como el tiempo de ejecución es una función lineal de la longitud, podemos decir que este método tiene un **tiempo lineal**. Comparado con el tiempo constante, es muy malo.

### 16.3. Cola enlazada

Podemos querer una implementación del TAD Cola que pueda realizar todas las operaciones en tiempo constante. Una manera de conseguir esto es implementar una **cola enlazada**, que es similar a una lista enlazada en el sentido de que está formada por cero o más objetos **NODO** enlazados. La diferencia es que la cola mantiene una referencia al primero y al último nodo simultáneamente, como se muestra en la figura.



Para la implementación enlazada vamos a cambiar la clase **NODO** para hacerla genérica, por lo que a partir de ahora se llamará **NODO[G]**. Lo único que hemos hecho ha sido parametrizar el tipo del dato contenido en **NODO**, que recordarás que se llamaba **carga**, con idea de que admita cualquier tipo de información. Por tanto, la clase **NODO[G]** quedaría así:

```
class NODO[G]

  creation
    make,
    make_par

  feature
    carga: G
    sgte: NODO[G]

    set_carga(c: G)
      do
        carga := c
      end

    set_sgte(s: NODO)
      do
        sgte := s
      end

    make is
      do
        carga := Void
        sgte := Void
      end
end
```

```

    make_par(elem: G; sig: NODO[G]) is
    do
        carga := elem
        sgte := sig
    end
end

```

Este es el aspecto que tiene una implementación de la cola enlazada usando objetos `NODO[G]`:

```

class COLA[G]

creation
    make

feature
    primero, ultimo: NODO[G]

    make is
    do
        primero := Void
        ultimo := Void
    end

    es_vacia: BOOLEAN is
    do
        Result := primero = Void
    end

    cabeza: G is
    require
        cola_no_vacia: not es_vacia
    do
        Result := primero.carga
    end
end

```

Hasta ahora la cosa está clara. En una cola vacía, tanto `primero` como `ultimo` son `Void`. Para comprobar si una lista está vacía, sólo tenemos que comprobar uno de los dos atributos.

Además, `cabeza` devuelve la información almacenada en el primer nodo de la lista (que también es el primero de la cola) pero para ello se requiere que la cola no esté vacía.

`insertar` es un poco más complicado porque tenemos que considerar varios casos especiales.

```

class COLA[G]
    -- ...
feature
    -- ...
    insertar(obj: G) is
    local
        nod: NODO[G]
    do
        create nod.make_par(obj, Void)
        if ultimo /= Void then

```

```

        ultimo.sgte := nod
    end
    ultimo := nod
    if primero = Void then
        primero := ultimo
    end
end
end

```

La primera condición comprueba que `ultimo` apunta a un nodo; si es así, entonces tenemos que crear una referencia al nuevo nodo.

La segunda condición trata el caso especial en el que la lista está inicialmente vacía. En este caso, tanto `primero` como `ultimo` deben apuntar al nuevo nodo.

`eliminar` también se ocupa de casos especiales.

```

class COLA[G]
  -- ...
feature
  -- ...
  eliminar is
  do
    if primero /= Void then
      primero := primero.sgte
    end
    if primero = Void then
      ultimo := Void
    end
  end
end
end

```

La primera condición comprueba si había algún nodo en la lista. Si es así, tenemos que copiar el nodo `sgte` en `primero`. La segunda condición trata el caso especial de que la lista esté vacía, en cuyo caso tenemos que hacer que `ultimo` apunte a `Void`.

Como ejercicio, dibuja diagramas que muestren ambas operaciones tanto en el caso normal como en los casos especiales, y convéncete de que son correctas.

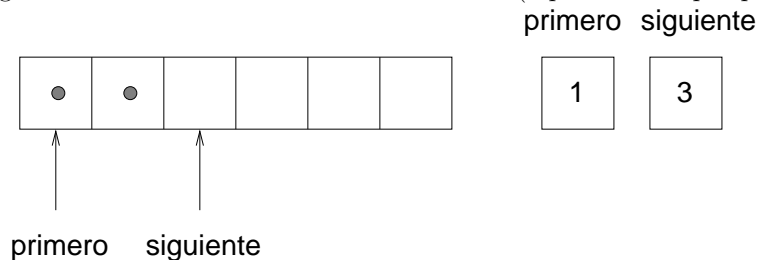
Claramente, esta implementación es más complicada que la implementación envoltorio, y es más difícil demostrar que es correcta. La ventaja es que hemos alcanzado el objetivo: tanto `insertar` como `eliminar` se ejecutan ahora en tiempo constante.

## 16.4. Buffer circular

Otra implementación típica de una cola es un **buffer circular**. “Buffer” es un término general que representa un espacio de almacenamiento temporal, aunque a menudo se corresponde con un array, como en este caso. Lo que hace que un buffer sea “circular” debería quedar aclarado en un minuto.

La implementación de un buffer circular es similar a la implementación de una pila basada en array, como en el apartado 15.8. Los elementos de la cola son almacenados en un array, y se usan índices para llevar la cuenta de dónde estamos dentro del array. En la implementación de la pila, sólo había un índice que apuntaba al último espacio utilizado. En la implementación de la cola existen dos índices: `primero` apunta al espacio en el array que contiene el primer consumidor en la fila, y `siguiente` apunta al siguiente espacio libre.

La figura siguiente muestra una cola con dos elementos (representados por puntos).



Hay dos formas de pensar en las variables **primero** y **siguiente**. Literalmente, son enteros, y sus valores se muestran en las cajas de la derecha. De una manera más abstracta, en cambio, son índices de un array, y por tanto a menudo se dibujan como flechas que apuntan a lugares dentro del array. Representarlos en forma de flecha es conveniente, pero debes recordar que los índices no son referencias; tan sólo son enteros.

Esta es una implementación incompleta de una cola basada en array:

```
class COLA[G]

creation make

feature
  vector: ARRAY[G]
  primero, siguiente: INTEGER

make is
do
  create vector.make(1, 128)
  primero := 1
  siguiente := 1
end

es_vacia: BOOLEAN is
do
  Result := primero = siguiente
end

end
```

Los atributos y el constructor están claros, aunque de nuevo tenemos un problema al tener que elegir un tamaño arbitrario para el array. Luego resolveremos ese problema, como hicimos con la pila, redimensionando el array cuando se llene.

La implementación de **es\_vacia** es un poco sorprendente. Debes pensar que **primero = 1** puede indicar una cola vacía, pero que descuida el hecho de que la cabeza de la cola no tiene que estar necesariamente al comienzo del array. En su lugar, sabemos que una cola está vacía si **primero** es igual a **siguiente**, en cuyo caso no quedan más elementos. Una vez que vamos la implementación de **insertar** y **eliminar**, esa situación cobrará más sentido.

```
class COLA[G]
  -- ...
feature
  -- ...
  insertar(elem: G) is
do
```

```

        vector.put(elem, siguiente)
        siguiente := siguiente + 1
    end

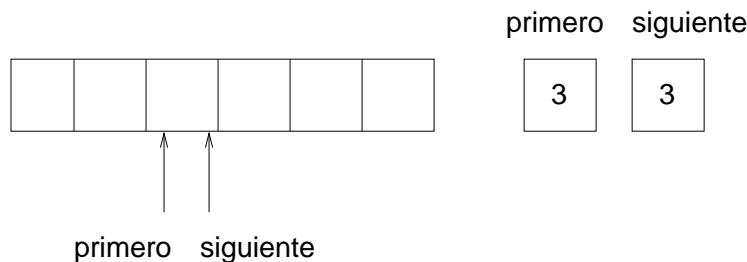
eliminar is
do
    primero := primero + 1
end

cabeza: G is
do
    Result := vector @ primero
end
end

```

**insertar** se parece mucho a **push** del apartado 15.8; coloca un nuevo elemento en el siguiente espacio libre disponible y luego incrementa el índice.

**eliminar** es similar. Incrementa **primero** para que haga referencia a la nueva cabeza de la cola. La siguiente figura muestra el aspecto de la cola después de que los dos elementos hayan sido eliminados.

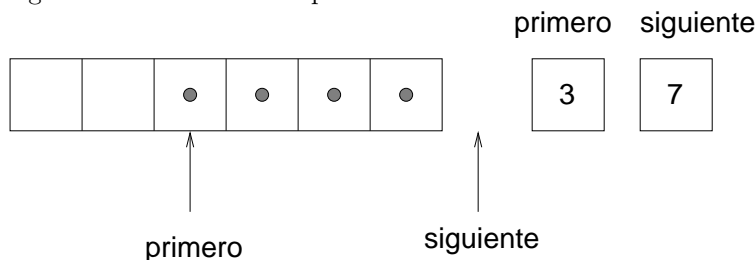


Siempre se cumple que **siguiente** apunta a un espacio libre. Si **primero** alcanza a **siguiente** y por tanto apunta al mismo espacio, entonces **primero** está haciendo referencia a un espacio “vacío”, y la cola por tanto está vacía. Pongo “vacío” entre comillas porque es posible que el espacio al que apunta **primero** contenga realmente algún valor (no hacemos nada para asegurarnos que los espacios vacíos contienen **Void**); por otra parte, como sabemos que la cola está vacía, nunca leeremos ese espacio, por lo que podemos pensar en él, de manera abstracta, como en un espacio vacío.

*Como ejercicio, arregla **eliminar** de forma que devuelva **Void** si la cola está vacía.*

El siguiente problema con esta implementación es que a veces se queda sin espacio. Cuando añadimos un elemento incrementamos **siguiente** y cuando eliminamos un elemento incrementamos **primero**, pero nunca decrementamos ninguno. ¿Qué ocurre si llegamos al final del array?

La siguiente figura muestra la cola después de añadir cuatro elementos más:



El array ahora está lleno. No hay un “siguiente espacio disponible”, por lo que **siguiente** no apunta a ningún sitio. Una posibilidad es que podemos redimensionar el array, como hicimos con la



implementación de la pila. Pero en ese caso el array se haría más y más grande independientemente de cuántos elementos hay realmente en la cola. Una solución mejor es saltar al comienzo del array y reutilizar el espacio disponible allí. Ese “salto” es la razón por la que a esta implementación se la denomina de buffer circular.

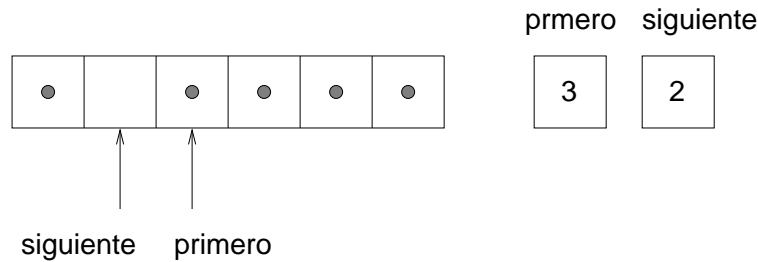
Una manera de hacer saltar el índice es añadir un caso especial cuando incrementamos un índice:

```
siguiente := siguiente + 1
if siguiente > vector.upper then
  siguiente := 1
end
```

Una alternativa mejor es usar el operador módulo:

```
siguiente := (siguiente + 1) \ vector.upper
```

Por otro lado, tenemos un último problema que resolver. ¿Cómo sabemos si la cola está *realmente* llena, en el sentido de que no podemos insertar otro elemento? La figura siguiente muestra el aspecto de la cola cuando está “llena”:



Todavía hay un espacio libre en el array, pero la cola está llena porque si insertamos otro elemento, entonces tenemos que incrementar **siguiente** de forma que **siguiente = primero**, ¡y en ese caso podría parecer que la cola está vacía!

Para evitar eso, sacrificamos un espacio en el array. Así que, ¿cómo decimos si la cola está llena?

```
if (siguiente + 1) \ vector.upper = primero then
```

¿Y qué debemos hacer si el array está lleno? En ese caso redimensionar el array es probablemente la única opción.

*Como ejercicio, junta todo el código de este apartado y escribe una implementación de una cola usando un buffer circular que se redimensione a sí misma cuando sea necesario.*

## 16.5. Cola con prioridad

El TAD Cola con prioridad tiene el mismo interface que el TAD Cola, pero distinta semántica. La interface es:

**make:** Crea una nueva cola vacía.

**insertar:** Añade un nuevo elemento a la cola.

**cabeza:** Devuelve un elemento de la cola. El elemento que se devuelve es aquél que tiene mayor prioridad.

**eliminar:** Elimina un elemento de la cola. El elemento que se elimina es aquél que tiene mayor prioridad.

**es\_vacia:** Comprueba si la cola está vacía.

La diferencia semántica es que el elemento que se elimina o se devuelve de la cola no es necesariamente el primero que se insertó. En su lugar, es el elemento que tiene más prioridad en la cola. Cuáles son las prioridades, y cómo se comparan unas con otras, no se especifican en la implementación de Cola con prioridad. Depende de cuáles son los elementos de la cola.

Por ejemplo, si los elementos de la cola tienen nombres, podríamos escogerlos en orden alfabético. Si son puntuaciones en el juego de los bolos, podríamos escogerlos de mayor a menor, pero si son puntuaciones en el golf, podríamos ir de menor a mayor.

Por tanto ahora nos enfrentamos a un nuevo problema. Queremos una implementación de Cola con prioridad que sea genérica —debe funcionar con cualquier tipo de objeto— pero al mismo tiempo el código que implementa a Cola con prioridad necesita tener la capacidad de comparar los objetos que contiene.

Hemos visto una manera de implementar estructuras de datos genéricas usando clases genéricas (con parámetros formales genéricos), pero eso no resuelve el problema, porque no hay forma de comparar objetos entre sí a menos que sepamos de qué tipo son.

La respuesta que ofrece Eiffel viene de la mano de dos nuevos conceptos, que son las **clases diferidas** y la **genericidad restringida**.

## 16.6. Clase diferida

Una clase diferida (o abstracta) es un conjunto de clases. La definición de una clase diferida especifica los requisitos que una clase debe satisfacer para ser un miembro de ese conjunto.

A menudo las clases diferidas tienen nombres que acaban con “able” para indicar la capacidad fundamental que la clase diferida requiere. Por ejemplo, todas las clases que proporcionan un método llamado `dibujar` pueden ser miembros de la clase diferida llamada `DIBUJABLE`.

Eiffel proporciona una clase diferida predefinida que puede usarse en la implementación de la Cola con prioridad. Se llama `COMPARABLE`, y significa precisamente eso. Toda clase que pertenezca a la clase diferida `COMPARABLE` debe proporcionar un método llamado `infix "<"` que compara dos objetos y devuelve `True` si uno es más pequeño, estrictamente, que el otro.

Muchas de las clases predefinidas de Eiffel son miembros de la clase diferida `COMPARABLE`.

En el siguiente apartado veremos cómo escribir un TAD que manipule una clase diferida. Después veremos cómo escribir una nueva clase que pertenezca a una clase diferida existente. Y después veremos cómo escribir una nueva clase diferida.

## 16.7. Implementación de Cola con prioridad basada en array

En la implementación de la Cola con prioridad, haremos lo mismo que con la implementación de la Cola simple: definiremos una clase genérica con un parámetro formal genérico que representará el tipo de los elementos de la Cola. Pero en este caso, sólo admitiremos como tipos aquellos que pertenezcan a la clase diferida `COMPARABLE`. Esto se representa así:

```
class COLA_PRIORIDAD[G -> COMPARABLE]
  -- ...
end
```

La sintaxis `[G ->COMPARABLE]` significa que `G` representa a cualquier posible tipo, *siempre que éste pertenezca a la clase `COMPARABLE`*. Es decir: estamos restringiendo el conjunto total de posibles tipos que podemos usar con `COLA_PRIORIDAD`, de manera que sólo admitiremos aquellos tipos que sean `COMPARABLES`. A ésto se le denomina **genericidad restringida**.

Por ejemplo, los atributos son:

```
class COLA_PRIORIDAD[G -> COMPARABLE]
  -- ...
feature
  vector: ARRAY[G]
  indice: INTEGER
  -- ...
end
```

Como es usual, `indice` es el índice del siguiente espacio libre en el array.

El constructor y `es_vacia` son similares a lo que hemos visto antes. Escogí de forma arbitraria el tamaño inicial del array.

```
class COLA_PRIORIDAD[G -> COMPARABLE]

creation make

feature
  vector: ARRAY[G]
  indice: INTEGER

  make is
    do
      create vector.make(1, 16)
      indice := 1
    end

  es_vacia: BOOLEAN is
    do
      Result := indice = 1
    end

  cabeza: G is
    do
      Result := vector @ indice
    end
end
```

`insertar` es similar a `push`:

```
class COLA_PRIORIDAD[G -> COMPARABLE]
  -- ...
feature
```

```

-- ...
insertar(elem: G) is
  do
    if indice > vector.upper then
      redimensionar
    end
    vector.put(elem, indice)
    indice := indice + 1
  end
end

```

He omitido la implementación de `redimensionar`. El único método sustancial en la clase es `eliminar`, que tiene que recorrer el array para encontrar y eliminar el elemento más grande:

```

class COLA_PRIORIDAD[G -> COMPARABLE]
  -- ...
  feature
    -- ...
    eliminar is
      local
        max_indice, i: INTEGER

      do
        if indice > 1 then
          max_indice := 1
          -- busca el índice del elemento con mayor prioridad
          from
            i := vector.lower
          until
            i > vector.upper
          loop
            if (vector @ i) < (vector @ max_indice) then
              max_indice := i
            end
            i := i + 1
          end
          indice := indice - 1
          vector.put(vector @ indice, max_indice)
        end
      end
    end
  end
end

```

A medida que recorremos el array, `max_indice` lleva la cuenta del índice del mayor elemento que hemos visto hasta ahora. Para saber qué elemento es el más grande, usamos la comparación `<`, que realmente supone una llamada al método infix `"<"`, de manera que `obj1 < obj2` se traduce por `obj1.infix "<"(obj2)`. Además, el tipo al que representa `G` seguro que tiene definido el método infix `"<"`, puesto que le obligamos a que sea `COMPARABLE`. Muchas clases predefinidas, como `INTEGER` o `STRING`, son comparables.

## 16.8. Un cliente de Cola con prioridad

La implementación de Cola con prioridad está escrita en términos de `G`, un nombre que representa a cualquier tipo `COMPARABLE`. Sin embargo, ¿no puedes crear directamente objetos `COMPARABLE`s! Adelante, intenta crear uno:

```

local
  comp: COMPARABLE
do
  create comp  -- ERROR
end

```

Obtendrás un mensaje del compilador que dice algo así como “Warning: Type COMPARABLE is deferred. Cannot create object.” (“Aviso: El tipo COMPARABLE es diferido. No se puede crear objeto”).

¿Por qué no pueden instanciarse clases diferidas? Porque una clase diferida sólo especifica requisitos (como tener un método `infix "<"`); no proporciona una implementación.

Para crear un objeto `COMPARABLE`, debes crear uno de los objetos que pertenecen al conjunto `COMPARABLE`, como `INTEGER`. Luego podrás usar ese objeto en cualquier lugar en el que se necesite un `COMPARABLE`.

```

local
  cp: COLA_PRIORIDAD[INTEGER]
  i: INTEGER
do
  create cp.make
  i := 17
  cp.insertar(i)
end

```

Este código crea una nueva Cola con prioridad vacía y un nuevo objeto `INTEGER`. Después inserta el objeto `INTEGER` en la cola. `insertar` espera un objeto de tipo `G` como parámetro, y ese `G` debe ser `COMPARABLE`, por lo que el método acepta perfectamente recibir un `INTEGER`. Si intentamos pasar un `RECTANGULO`, que no pertenece a `COMPARABLE`, obtendremos un mensaje de error indicando que el tipo no cumple la restricción de ser `COMPARABLE`.

Para sacar los elementos de la cola, tenemos que invertir el proceso:

```

local
  cp: COLA_PRIORIDAD[INTEGER]
  elem: INTEGER
do
  -- ...
  from
  until
    cp.es_vacia
  loop
    elem := cp.cabeza
    print(elem) print("%N")
    cp.eliminar
  end
end

```

Este bucle elimina todos los elementos de la cola a medida que los va imprimiendo.

## 16.9. La clase GOLFISTA

Finalmente, veremos cómo podemos crear una nueva clase que pertenezca a `COMPARABLE`. Como un ejemplo como usual de definición de prioridad “alta”, usaremos golfistas:

```

class GOLFISTA

inherit
  COMPARABLE

creation make

feature
  nombre: STRING
  puntos: INTEGER

  make(n: STRING; p: INTEGER) is
    do
      nombre := n
      puntos := p
    end
end

```

La definición de clase y el constructor son como los de siempre; la diferencia es que tenemos que declarar que `GOLFISTA` “hereda” de `COMPARABLE`, y de esta forma se convierte él mismo en un tipo `COMPARABLE`.

Para poder usar la nueva clase, definimos una clase raíz (y su método raíz):

```

class RAIZ

creation make

feature
  make is
    local
      seve: GOLFISTA
    do
      create seve.make("Severiano Ballesteros", 61)
    end

  to_string: STRING is
    do
      Result := nombre + " " + puntos.to_string
    end
end

```

Si intentas compilar el programa en este momento, obtendrás algo como “Warning: < is a deferred feature in GOLFISTA. [...] infix "<" (other: like Current): BOOLEAN is”. En otras palabras, para que `GOLFISTA` pueda ser miembro de `COMPARABLE`, tiene que proporcionar un método llamado `infix "<"` con la interface que se te indica. Así que escribámoslo:

```

class GOLFISTA

inherit
  COMPARABLE
  redefine
    infix "<"
end

```

```

-- ...
feature
  -- ...
  infix "<"(other: like Current): BOOLEAN is
    do
      -- para los golfistas, menos es mejor
      Result := Current.puntos < other.puntos
    end
end

```

Aquí hay dos cosas un poco sorprendentes. Primero, en la cláusula `inherit` decimos que heredamos de la clase `COMPARABLE`, pero además dejamos claro que vamos a (re)definir el comportamiento del método `infix "<"` de la clase `COMPARABLE`. Esto es necesario a veces, pero siempre interesante desde el punto de vista de la documentación del código.

Segundo, el parámetro está definido como `like Current`. Con esto se está diciendo, explícitamente, que el tipo del parámetro tiene que ser exactamente el mismo que el del objeto actual (evidentemente, porque si no, no tendría sentido compararlos), que en este caso es `GOLFISTA`. ¿Por qué no usamos directamente `GOLFISTA` en lugar de `like Current`, ya que significan lo mismo? Simplemente, porque el método `infix "<"` debe estar definido exactamente como se nos pide. Es algo que se nos viene impuesto desde la clase `COMPARABLE`, y debemos cumplirlo si queremos que `GOLFISTA` pertenezca a `COMPARABLE`.

Finalmente, podemos crear algunos golfistas:

```

class RAIZ
  creation make

  feature
    make is
      local
        tiger, phil, hal: GOLFISTA
      do
        create tiger.make("Tiger Woods", 61)
        create phil.make("Phil Mickelson", 72)
        create hal.make("Hal Sutton", 69)
      end
    end
end

```

Y ponerlos en la cola:

```

class RAIZ
  -- ...
  feature
    make is
      local
        tiger, phil, hal: GOLFISTA
        cp: COLA_PRIORIDAD[GOLFISTA]
      do
        -- creamos los golfistas, y luego...
        cp.insertar(tiger)
        cp.insertar(phil)
        cp.insertar(hal)
      end
    end
end

```

Cuando los sacamos:

```

from
until
    cp.es_vacia
loop
    print(cp.cabeza.to_string)
    print("%N")
    cp.eliminar
end

```

Aparecen en orden descendente (para los golfistas):

Tiger Woods	61
Hal Sutton	69
Phil Mickelson	72

Cuando cambiamos de INTEGERS a GOLFISTAS, no hemos tenido que hacer ningún cambio en COLA\_PRIORIDAD. Con lo cual tenemos éxito en nuestro objetivo de mantener una barrera entre COLA\_PRIORIDAD y las clases que la utilizan, permitiendo la reutilización del código sin tener que modificarlo. Más aún, hemos sido capaces de dar al código cliente el control sobre la definición de infix "<", haciendo que la implementación de COLA\_PRIORIDAD sea aún más versátil.

## 16.10. Glosario

**cola:** Un conjunto ordenado de objetos que esperan un algún tipo de servicio.

**disciplina de encolado:** Las reglas que determinan qué miembro de una cola pasará a continuación.

**FIFO:** “first in, first out,” una disciplina de encolado en la cual el primer miembro en llegar es el primero en ser servido.

**cola con prioridad:** Una disciplina de encolado en la cual cada miembro tiene una prioridad determinada por factores externos. El miembro con mayor prioridad es el primero en ser servido.

**Cola con prioridad:** Un TAD que define las operaciones que pueden realizarse sobre una cola con prioridad.

**envoltorio:** Una definición de clase que implementa un TAD con definiciones de métodos que son llamadas a otros métodos, quizás con transformaciones sencillas. El envoltorio no hace ningún trabajo significativo, pero mejora o estandariza el interface que ve el cliente.

**riesgo en eficiencia:** Un peligro asociado a un envoltorio por el cual algunos de sus métodos podrían ser implementados ineficientemente de una manera que no resulta aparente para el cliente.

**tiempo constante:** Una operación cuyo tiempo de ejecución no depende del tamaño de la estructura de datos.

**tiempo lineal:** Una operación cuyo tiempo de ejecución es una función lineal del tamaño de la estructura de datos.

**cola enlazada:** Una implementación de una cola que utiliza una lista enlazada y referencias al primer y último nodos.



**buffer circular:** Una implementación de una cola que utiliza un array e índices para el primer elemento y el siguiente espacio disponible.

**clase diferida:** Un conjunto de clases. La especificación de una clase diferida enumera los requisitos que una clase debe satisfacer para ser incluida en el conjunto.

**genericidad restringida:** Una característica de Eiffel que permite limitar el conjunto de posibles tipos representados por un parámetro formal genérico a aquellos tipos que sean descendientes de una determinada clase.

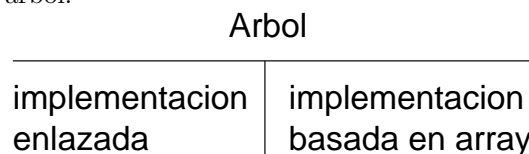
## Capítulo 17

# Árboles

Este capítulo presenta una nueva estructura de datos llamada árbol, algunos de sus usos y dos formas de implementarla.

Una posible fuente de confusión es la distinción entre un TAD, una estructura de datos, y una implementación de un TAD o una estructura de datos. No existe una respuesta universal, porque algo que es un TAD a un cierto nivel podría convertirse en la implementación de otro TAD.

Para ayudar a tener las cosas claras, a menudo es útil dibujar un diagrama que muestra la relación entre un TAD y sus posibles implementaciones. Esta figura muestra que existen dos implementaciones de un árbol:



La línea horizontal de la figura representa la frontera de abstracción entre el TAD y sus implementaciones.

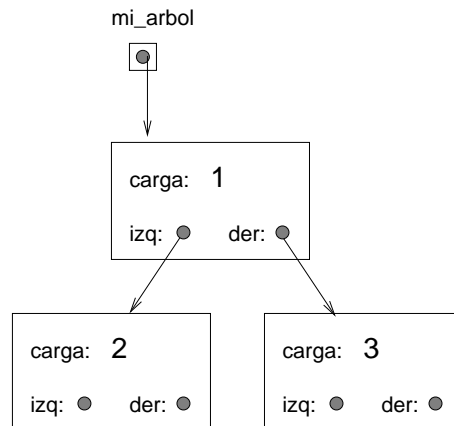
### 17.1. Un nodo del árbol

Como las listas, los árboles están hechos de nodos. Un tipo habitual de árbol es el **árbol binario**, en el que cada nodo contiene referencias a otros dos nodos (posiblemente `Void`). La definición de clase tiene el siguiente aspecto:

```
class ARBOL[G]

feature
  carga: G
  izq, der: ARBOL[G]
end
```

Como los nodos de una lista, los nodos de los árboles contienen información: en este caso, un dato cuyo tipo, `G`, es genérico, y que coincide con el parámetro formal genérico de la clase. Los otros atributos son `izq` y `der`, en concordancia con la manera estándar de representar árboles gráficamente:



La parte superior del árbol (el nodo al que se hace referencia con `mi_arbol`) se llama la **raíz**. Siguiendo con la metáfora de los árboles, los otros nodos se llaman ramas, y los nodos del fondo, que contienen referencias a `Void`, se llaman **hojas**. Puede parecer extraño que dibujemos el árbol con la raíz arriba y las hojas abajo, pero eso no es lo más extraño.

Para poner las cosas aún peor, los informáticos utilizan otra metáfora: la del árbol genealógico. El nodo superior a menudo es llamado el **padre** y los nodos a los que apunta son sus **hijos**. Los nodos con el mismo padre se llaman **hermanos**, y así.

Finalmente, existe además un vocabulario geométrico al hablar de árboles. Ya mencioné anteriormente la izquierda y la derecha, pero además existe “arriba” (hacia el padre/raíz) y abajo (hacia los hijos/hojas). Además, todos los nodos que están a la misma distancia del raíz forman un **nivel** del árbol.

No sé por qué necesitamos tres metáforas para hablar de árboles, pero ahí están.

## 17.2. Construir árboles

El proceso de ensamblar nodos de árboles es similar al proceso de ensamblar listas. Tenemos un constructor para nodos de árboles que inicializa a los atributos.

```

class ARBOL[G]

  creation make

  feature
    -- ...
    make(c: G; i, d: ARBOL[G]) is
      do
        carga := c
        izq := i
        der := d
      end
    end
end

```

Creamos los nodos hijos primero:

```

class RAIZ

  creation make

```

```

feature
  make is
    local
      izquierda, derecha, mi_arbol: ARBOL[INTEGER]
    do
      create izquierda.make(2, Void, Void)
      create derecha.make(3, Void, Void)
    end
end

```

Podemos crear el nodo padre y enlazarlo con los hijos al mismo tiempo:

```

make is
  local
    izquierda, derecha, mi_arbol: ARBOL[INTEGER]
  do
    create izquierda.make(2, Void, Void)
    create derecha.make(3, Void, Void)
    create mi_arbol.make(1, izquierda, derecha)
  end
end

```

Este código produce el estado que muestra la figura anterior.

### 17.3. Recorrer árboles

A partir de ahora, cada vez que veas una nueva estructura de datos, tu primera pregunta podría ser “¿Cómo puedo recorrerla?”. La forma más natural de recorrer un árbol es recursivamente. Por ejemplo, para sumar todos los enteros de un árbol, podemos escribir este método externo:

```

class RAIZ
  -- ...
feature
  -- ...
  total(ar: ARBOL[INTEGER]): INTEGER is
    local
      carga: INTEGER
    do
      if ar = Void then
        Result := 0
      else
        carga := ar.carga
        Result := carga + total(ar.izq) + total(ar.der)
      end
    end
end
end

```

Este es un método externo porque queremos usar `Void` para representar el árbol vacío, y hacer que sea el caso base de la recursión. Si el árbol está vacío, el método devuelve 0. En otro caso hace dos llamadas recursivas para calcular el valor total de sus dos hijos. Finalmente, le añade su propia carga y devuelve el total.

Aunque este método funciona, resulta algo difícil incluirlo en un diseño orientado a objetos. No debería aparecer en la clase `ARBOL` porque requiere que la carga sea un objeto `INTEGER`. Si hacemos esa suposición en `ARBOL` entonces perdemos las ventajas de una estructura de datos genérica.

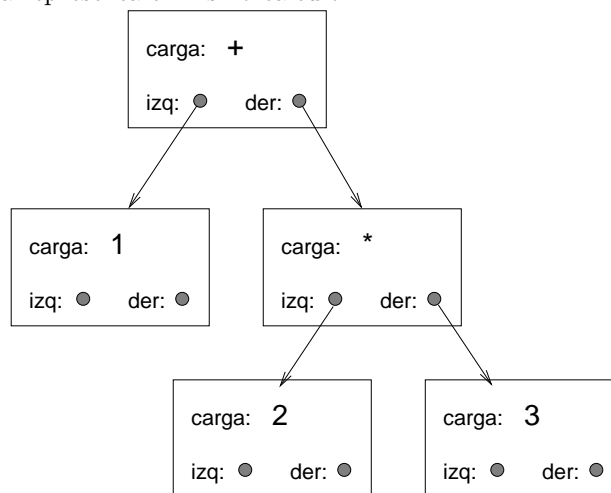
Por otra parte, este código accede a los atributos de los nodos de `ARBOL`, por lo que “conoce” más de lo que debería acerca de la implementación del árbol. Si cambiamos esa implementación más tarde (y lo haremos) este código podría dejar de funcionar.

A lo largo de este capítulo desarrollaremos maneras de resolver este problema, permitiendo al código cliente recorrer árboles que contengan cualquier tipo de objetos sin romper la frontera de abstracción entre el código cliente y la implementación. Antes de eso, veamos una aplicación de los árboles.

## 17.4. Árboles de expresiones

Un árbol es una forma natural de representar la estructura de una expresión. A diferencia de otras notaciones, puede representar el cálculo de forma no ambigua. Por ejemplo, la expresión infija  $1 + 2 * 3$  es ambigua a menos que sepamos que el producto va antes que la suma.

La siguiente figura representa el mismo cálculo:



Los nodos pueden ser operandos como 1 y 2 u operadores como + y \*. Los operandos son nodos hoja; los nodos operador contienen referencias a sus operandos (todos esos operandos son **binarios**, lo que significa que tienen exactamente dos operandos).

Mirando esta figura, no hay problema acerca de cuál es el orden de las operaciones: el producto se hace primero para calcular el primer operando de la suma.

Los árboles de expresiones como éste tienen muchos usos. El ejemplo que vamos a ver es el de traducir de un formato (postfijo) a otro (infijo). Árboles similares se usan dentro de los compiladores para analizar, optimizar y traducir programas.

## 17.5. Recorrido

Ya he mencionado que la recursividad proporciona una forma natural de recorrer un árbol. Podemos imprimir el contenido de un árbol de expresión así:

```
class RAIZ
-- ...
```

```

feature
  -- ...
  imprimir(ar: ARBOL[G]) is
    do
      if ar /= Void then
        print(ar.carga + " ")
        imprimir(ar.izq)
        imprimir(ar.der)
      end
    end
end

```

En otras palabras, para imprimir un árbol, primero imprimimos el contenido de la raíz, luego imprimimos el subárbol izquierdo completo, y luego imprimimos el subárbol derecho completo. Esta forma de recorrer un árbol se denomina recorrido en **preorden**, porque el contenido de la raíz aparece antes que el contenido de los hijos.

Para la expresión del ejemplo la salida es  $+ 1 * 2 3$ . Esto no es postfijo ni infijo; es una nueva notación llamada **prefija**, en la cual los operadores aparecen antes que los operandos.

Puedes sospechar que si recorremos el árbol en un orden diferente obtendremos la expresión en una notación diferente. Por ejemplo, si imprimimos los subárboles primero, y luego el nodo raíz:

```

class RAIZ
  -- ...
feature
  -- ...
  imprimir_postorden(ar: ARBOL[G]) is
    do
      if ar /= Void then
        imprimir_postorden(ar.izq)
        imprimir_postorden(ar.der)
        print(ar.carga + " ")
      end
    end
end

```

¡Obtenemos la expresión en postfijo ( $1 2 3 * +$ )! Como el nombre del método anterior indica, este orden de recorrido se denomina recorrido en **postorden**. Finalmente, para recorrer un árbol en **inorden**, imprimimos el árbol izquierdo, después la raíz, y por último el árbol derecho:

```

class RAIZ
  -- ...
feature
  -- ...
  imprimir_inorden(ar: ARBOL[G]) is
    do
      if ar /= Void then
        imprimir_inorden(ar.izq)
        print(ar.carga + " ")
        imprimir_inorden(ar.der)
      end
    end
end

```

El resultado es  $1 + 2 * 3$ , que es la expresión en infijo.

Para ser sincero, tengo que admitir que he omitido una complicación importante. A veces cuando escribimos una expresión en notación infija tenemos que usar paréntesis para preservar el orden de las operaciones. Por lo que un recorrido en inorden no es suficiente para generar una expresión infija.

En cualquier caso, con unas cuantas mejoras, el árbol de expresión y los tres recorridos recursivos proporcionan una manera general de traducir expresiones de un formato a otro.

## 17.6. Encapsulación

Como mencioné antes, hay un problema con la forma en la que hemos estado recorriendo árboles: rompe la frontera entre el código cliente (la aplicación que usa el árbol) y el código proveedor (la implementación `ARBOL`). Idealmente, el código del árbol debería ser general; no debería saber nada sobre árboles de expresión. Y el código que genera y recorre el árbol de expresión no debería saber nada acerca de la implementación de los árboles. Este criterio de diseño se denomina **encapsulación de objeto** para distinguirlo de la encapsulación que ya vimos en el apartado 6.6, que podríamos denominar **encapsulación de método**.

En la versión actual, el código de `ARBOL` conoce demasiado al cliente. En su lugar, la clase `ARBOL` debería proporcionar la capacidad general de recorrer un árbol de varias formas. Durante el recorrido, debería realizar sobre los nodos las operaciones especificadas por el cliente.

Para facilitar esta separación de intereses, crearemos una nueva clase diferida, llamada `VISITABLE`. Se obligará a que los elementos almacenados en un árbol sean visitables, lo que significa que deben definir un método llamado `visitar` que haga lo que el cliente quiere que se haga con cada nodo. De esta forma el árbol puede realizar el recorrido y el cliente puede realizar las operaciones con los nodos.

Estos son los pasos que tenemos que realizar para incluir una clase diferida entre un cliente y un proveedor:

1. Definir una clase diferida que especifique los métodos que el código proveedor necesita invocar sobre sus componentes.
2. Escribir el código proveedor en términos de la nueva clase diferida, usando un parámetro formal genérico que restrinja el conjunto de tipos posibles a aquellos que sean herederos de esa clase diferida.
3. Definir una clase efectiva que pertenezca a la clase diferida y que implemente los métodos requeridos de forma apropiada para el cliente.
4. Escribir el código cliente para usar la nueva clase efectiva.

Los siguientes apartados muestran estos pasos.

## 17.7. Definir una clase diferida

Una definición de una clase diferida se parece mucho a una definición de una clase efectiva, con estas dos excepciones:

1. Se utilizan las palabras clave `deferred class` en lugar de `class`.
2. En cada método interno, en lugar de la implementación aparece la palabra clave `deferred`.

La definición de VISITABLE es

```
deferred class VISITABLE

  feature
    visitar is
      deferred
    end
  end
end
```

¡Eso es! La palabra **deferred** es la palabra clave Eiffel para una clase diferida. La definición de **visitar** se parece a la de cualquier otro método, excepto que no tiene cuerpo, y en su lugar vuelve a aparecer la palabra **deferred**. Esta definición especifica que toda clase efectiva que sea heredera de **VISITABLE** debe tener un método llamado **visitar** que no recibe parámetros y no devuelve nada. Un método definido de esta forma se denomina **método diferido**.

Como otras definiciones de clase, las definiciones de clases diferidas van en un fichero aparte, cuyo nombre es el nombre de la clase (en este caso **visitable.e**).

## 17.8. Hacer efectiva una clase diferida

Si estamos usando un árbol de expresión para generar notación infija, entonces “visitar” un nodo significa imprimir su contenido. Como el contenido de una expresión son tokens, crearemos una nueva clase efectiva llamada **TOKEN** que herede de la clase **VISITABLE** y redefina los métodos diferidos de ésta última.

```
class TOKEN

  inherit
    VISITABLE
    redefine
      visitar
    end

  creation make

  feature
    str: STRING

    make(st: STRING) is
      do
        str := st
      end

    visitar is
      do
        print(str + " ")
      end
    end
  end
end
```

Cuando compilamos esta definición de clase (que estará en un fichero llamado **token.e**), el compilador comprueba que los métodos proporcionados satisfacen los requisitos especificados por la clase diferida. Si no, producirá un mensaje de error.



El siguiente paso es modificar el analizador para poner objetos `TOKEN` en el árbol en lugar de `STRINGS`.

*Como ejercicio, escribe una versión de `imprimir_preorden` llamada `visitar_preorden` que recorre un árbol y llama a `visitar` sobre cada nodo en preorden.*

## 17.9. Implementación de árboles basada en array

¿Qué significa “implementar” un árbol? Hasta ahora sólo hemos visto una implementación de un árbol, una estructura de datos enlazada similar a una lista enlazada. Pero hay otras estructuras que nos gustaría identificar como árboles. Todo aquello que pueda ejecutar el conjunto básico de operaciones sobre árboles debería ser reconocido como árbol.

Y, ¿cuáles son las operaciones de un árbol? En otras palabras, ¿cómo definimos el TAD `Arbol`?

**make:** Crea un árbol vacío.

**es\_vacio:** ¿Es este árbol el árbol vacío?

**izq:** Devuelve el hijo izquierdo de este nodo, o un árbol vacío si no tiene.

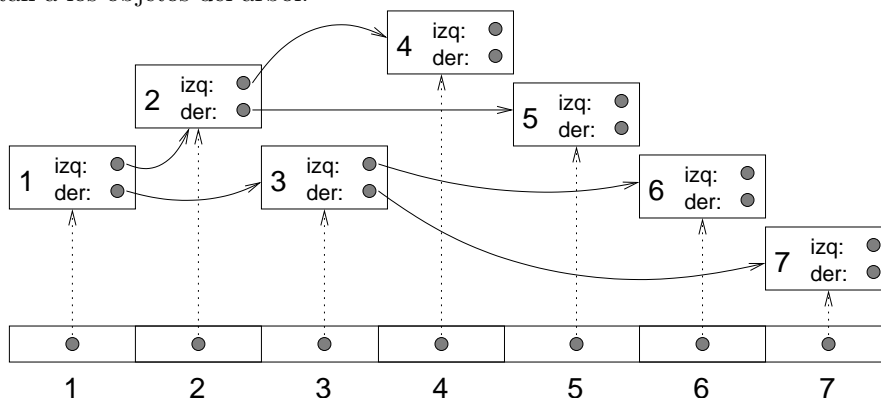
**der:** Devuelve el hijo derecho de este nodo, o un árbol vacío si no tiene.

**padre:** Devuelve el padre de este nodo, o un árbol vacío si este nodo es la raíz.

En la implementación que hemos visto, el árbol vacío se representa por el valor especial `Void`. `izq` y `der` se realizan accediendo a los atributos del nodo. Todavía no hemos implementado `padre` (puedes pensar cómo hacerlo).

Existe otra implementación de árboles que utiliza arrays e índices en lugar de objetos y referencias. Para ver cómo funciona, empezaremos viendo una implementación híbrida que utiliza al mismo tiempo arrays y objetos.

La figura muestra un árbol como los que hemos visto hasta ahora, aunque tumbado de lado, con la raíz a la izquierda y las hojas a la derecha. En la parte inferior hay un array de referencias que apuntan a los objetos del árbol.



En este árbol la carga de cada nodo es el índice del nodo dentro del array, pero por supuesto esto no es cierto en general.

Ahora tenemos un árbol donde cada nodo tiene un índice único. Más aún, los índices han sido asignados a los nodos de acuerdo a un patrón deliberado, con idea de alcanzar los siguientes resultados:

1. El hijo izquierdo del nodo cuyo índice es  $i$  tiene el índice  $2i$ .
2. El hijo derecho del nodo cuyo índice es  $i$  tiene el índice  $2i + 1$ .
3. El padre del nodo cuyo índice es  $i$  tiene el índice  $i/2$  (redondeado a la baja).

Usando estas fórmulas, podemos implementar `izq`, `der` y `padre` simplemente haciendo cálculos aritméticos; ¡no tenemos que usar ninguna referencia!

Como no usamos las referencias, podemos olvidarnos de ellas, lo que supone que desde el punto de vista de la utilización que le estamos dando, un nodo es sólo carga, y nada más. Esto significa que podemos implementar el árbol como un array de objetos carga; no necesitamos nodos de árboles en absoluto.

Este es el aspecto de una implementación:

```
class ARBOL[G]

  creation make

  feature
    vector: ARRAY[G]

    make is
      do
        create vector.make(1, 128)
      end
    end

end
```

Ninguna sorpresa hasta ahora. El atributo es un array de Gs. El constructor inicializa este array con un tamaño inicial arbitrario (siempre podemos redimensionarlo más tarde).

Para comprobar si un árbol está vacío, podemos comprobar si el nodo raíz es `Void`. De nuevo, el nodo raíz se localiza en el índice 1.

```
class ARBOL[G]
  -- ...
  feature
    -- ...
    es_vacio: BOOLEAN is
      do
        Result := (vector @ 1) = Void
      end
    end
  end
end
```

La implementación de `izq`, `der` y `padre` es sólo aritmética:

```
class ARBOL[G]
  -- ...
  feature
    -- ...
    izq(i: INTEGER): INTEGER is
      do
        Result := 2 * i
      end
    end
  end
end
```

```

    der(i: INTEGER): INTEGER is
    do
        Result := 2 * i + 1
    end

    padre(i: INTEGER): INTEGER is
    do
        Result := i // 2
    end
end

```

Tan sólo resta un problema. Las “referencias” a nodos que ahora tenemos no son realmente referencias; son índices enteros. Para acceder a la carga en sí misma, tenemos que asignar o acceder a un elemento del array. Para esa clase de operación, a menudo es una buena idea proporcionar métodos que realicen un chequeo simple contra errores antes de acceder a la estructura de datos. Además, en Eiffel no es posible asignar un valor a un atributo de una clase desde otra clase, así que en nuestro caso el método `set_carga` es obligatorio.

```

class ARBOL[G]
-- ...
feature
-- ...
    get_carga(i: INTEGER): G is
    do
        if i < vector.lower or i > vector.upper then
            Result := Void
        else
            Result := vector @ i
        end
    end

    set_carga(i: INTEGER; elem: G) is
    do
        if i >= vector.lower and i <= vector.upper then
            vector.put(elem, i)
        end
    end
end

```

Los métodos como éstos a menudo se denominan **métodos de acceso** porque proporcionan acceso a una estructura de datos (la capacidad de devolver y asignar elementos) sin dejar que el cliente vea los detalles de la implementación.

Finalmente, estamos preparados para construir un árbol. En otra clase (el cliente), podemos escribir

```

class RAIZ

creation make

feature
    make is
    local
        ar: ARBOL[STRING]

```

```

        raiz: INTEGER
    do
        create ar.make
        raiz := 1
        ar.set_carga(raiz, "carga para raiz")
    end
end

```

El constructor crea un árbol vacío. En este caso suponemos que el cliente sabe que el índice de la raíz es 1 aunque sería preferible que la implementación proporcionara esa información. En todo caso, invocar a `set_carga` mete la cadena "carga para raiz" dentro del nodo raíz.

Para añadir hijos al nodo raíz:

```

class RAIZ
  -- ...
feature
  -- ...
  make is
    local
      ar: ARBOL[STRING]
      raiz: INTEGER
    do
      ar.set_carga(ar.izq(raiz), "carga para izq")
      ar.set_carga(ar.der(raiz), "carga para der")
    end
  end
end

```

En la clase `ARBOL` podemos proporcionar un método que imprima el contenido del árbol en preorden.

```

class ARBOL[G]
  -- ...
feature
  -- ...
  imprimir_preorden(i: INTEGER) is
    do
      if get_carga(i) /= Void then
        print((get_carga(i)).to_string)
        imprimir_preorden(izq(i))
        imprimir_preorden(der(i))
      end
    end
  end
end

```

Invocamos a este método desde el cliente pasándole la raíz como parámetro.

```

ar.imprimir_preorden(raiz)

```

La salida es

```

carga para raiz
carga para izq
carga para der

```

Esta implementación proporciona las operaciones básicas requeridas para ser un árbol, pero deja mucho que desear. Como ya dije antes, necesitamos que el cliente conozca mucha información sobre la implementación, y la interface que el cliente ve, con índices y todo eso, no es muy agradable.

Además, tenemos el típico problema de la implementación con arrays, o sea, que el tamaño inicial del array es arbitrario y puede necesitar una redimensión. El último problema puede ser resuelto utilizando los métodos predefinidos de la clase `ARRAY`.

## 17.10. Diseño por Contrato

La implementación anterior puede mejorarse añadiendo al código de los métodos sus correspondientes precondiciones y postcondiciones. El resultado podría ser algo así:

```
class ARBOL[G]

  creation make

  feature
    vector: ARRAY[G]

    make is
      do
        create vector.make(1, 128)
      end

    es_vacio: BOOLEAN is
      do
        Result := (vector @ 1) = Void
      ensure
        ((vector @ 1) = Void) implies (Result = True)
      end

    izq(i: INTEGER): INTEGER is
      require
        i >= vector.lower and i <= vector.upper
      do
        Result := 2 * i
      end

    der(i: INTEGER): INTEGER is
      require
        i >= vector.lower and i <= vector.upper
      do
        Result := 2 * i + 1
      end

    padre(i: INTEGER): INTEGER is
      require
        i >= vector.lower and i <= vector.upper
      do
        Result := i // 2
      end
  end
```

```

get_carga (i: INTEGER): G is
do
  if i < vector.lower or i > vector.upper then
    Result := Void
  else
    Result := vector @ i
  end
end

set_carga(i: INTEGER; elem: G) is
require
  i >= vector.lower
  i <= vector.upper
do
  vector.put(elem, i)
end
end

```

Observa que en el método `set_carga` hemos reemplazado la sentencia alternativa por precondiciones. Antes, cuando no se cumplía la condición, el método no hacía nada. Ahora, si no se cumple la condición, simplemente no funcionará (provocará una excepción). La ventaja fundamental es que las precondiciones forman parte de la documentación del método, y por tanto quedan explícitas al cliente del método, cosa que no ocurre con las sentencias que forman el método.

### 17.11. La clase ARRAY

La clase `ARRAY` de Eiffel es mucho más versátil de lo que parece. Una de sus grandes ventajas es que permite que los arrays cambien de tamaño en tiempo de ejecución, para adecuarse a las necesidades concretas del cliente.

Los métodos `add_first` y `add_last`, entre otros, sirven para insertar un nuevo elemento en el array, incrementando su tamaño anterior en una unidad. Por contra, el método `put` asigna un valor a una posición del array, sin modificar su tamaño. Finalmente, el método `resize` cambia el tamaño del array, alterando los índices inferior y superior del mismo.

Como el código cliente no tiene acceso a la implementación de un array, no está claro cómo debemos recorrerlo. Por supuesto, una posibilidad es utilizar una variable contador como índice del vector:

```

local
  i: INTEGER
  a: ARRAY[INTEGER]
do
  from
    i := a.lower
  until
    i > a.upper
  loop
    print a @ i
    i := i + 1
  end
end
end

```

No hay nada incorrecto aquí, pero existe otra manera que sirve como demostración de las capacidades de la clase `ITERATOR`. Los arrays proporcionan una característica llamada `get_new_iterator` que devuelve un objeto `ITERATOR[G]`, el cual hace posible recorrer el array.

## 17.12. La clase `ITERATOR`

`ITERATOR` es una clase diferida que se encuentra predefinida en Eiffel. Especifica cuatro métodos:

**start:** Posiciona el iterador en el primer objeto a ser recorrido en la colección.

**is\_off:** ¿Hemos llegado al final de la secuencia?

**item:** Devuelve el objeto situado en la posición actual en la secuencia.

**next:** Posiciona el iterador en el siguiente objeto de la secuencia.

El siguiente ejemplo utiliza un iterador para recorrer e imprimir los elementos de un array.

```

local
  ar: ARRAY[INTEGER]
  it: ITERATOR[INTEGER]
do
  -- crear e inicializar el array,
  -- y después...
  from
    it := ar.get_new_iterator
    it.start
  until
    it.is_off
  loop
    print(it.item) print("%N")
    it.next
  end
end
end

```

Una vez que se ha creado el `ITERATOR`, se convierte en un objeto separado del array original. Cambios posteriores en el array no se reflejarán en el iterador. De hecho, si cambias el array después de crear el iterador, el iterador dejará de ser válido.

En un apartado anterior usamos la clase diferida `VISITABLE` para permitir a los clientes recorrer una estructura de datos sin tener que conocer los detalles de su implementación. Los iteradores proporcionan otra forma de hacer lo mismo. En el primer caso, el proveedor realiza la iteración y llama al código cliente para “visitar” cada elemento. En el segundo caso el proveedor le da al cliente un objeto que éste puede usar para seleccionar los elementos de uno en uno (aunque en un orden controlado por el proveedor).

*Como ejercicio, escribe una clase efectiva llamada `PRE_ITERADOR` que herede de la clase `ITERATOR`, y escribe un método llamado `iterador_preorden` para la clase `ARBOL` que devuelva un `PRE_ITERADOR` que seleccione los elementos del árbol en preorden. Recuerda que `PRE_ITERADOR` debe (re)definir todos los métodos diferidos de `ITERADOR`, que en este caso son todos.*

### 17.13. Glosario

**árbol binario:** Un árbol en el que cada nodo apunta a 0, 1 ó 2 nodos dependientes.

**raíz:** El nodo más alto de un árbol, al que no se refiere ningún otro nodo.

**hoja:** El nodo (o nodos) más bajo de un árbol, el cual no apunta a ningún otro nodo.

**padre:** El nodo que apunta a uno dado.

**hijo:** Uno de los nodos referenciados por un nodo.

**nivel:** El conjunto de los nodos que se encuentran a la misma distancia de la raíz.

**notación prefija:** Una manera de escribir una expresión matemática en la que cada operador aparece antes que sus operandos.

**preorden:** Una manera de recorrer un árbol, visitando cada nodo antes que a sus hijos.

**postorden:** Una manera de recorrer un árbol, visitando los hijos de cada nodo antes que al nodo mismo.

**inorden:** Una manera de recorrer un árbol, visitando el subárbol izquierdo, después la raíz, y después el subárbol derecho.

**operador binario:** Un operador que necesita dos operandos.

**encapsulación de objeto:** El criterio de diseño de mantener las implementaciones de dos objetos tan separadas como sea posible. Ninguna clase debería conocer los detalles de implementación de la otra.

**encapsulación de método:** El criterio de diseño de mantener la interface de un método separado de los detalles de su implementación.

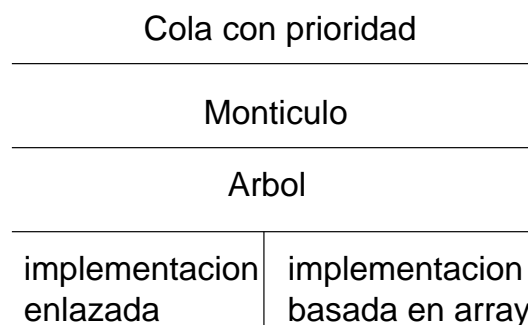


## Capítulo 18

# Montículos

### 18.1. El Montículo

Un montículo es un tipo especial de árbol que resulta ser una implementación eficiente de una cola con prioridad. Esta figura muestra las relaciones entre las estructuras de datos de este capítulo.



Normalmente intentamos mantener la mayor distancia posible entre un TAD y su implementación, pero en el caso del Montículo, romperemos un poco esa frontera. La razón es que estamos interesados en la eficiencia de las operaciones que implementamos. Para cada implementación hay varias operaciones que son eficientes y fáciles de implementar, y otras que son difíciles y lentas.

La implementación de un árbol basada en array funciona particularmente bien como implementación de un Montículo. Las operaciones que el array realiza bien son exactamente las operaciones que necesitamos para implementar un Montículo.

Para entender esta relación, procederemos en varios pasos. Primero, necesitamos desarrollar formas de comparar la eficiencia de varias implementaciones. Luego, veremos la eficiencia de las operaciones sobre Montículos. Finalmente, compararemos la implementación basada en Montículo de una Cola con prioridad con el resto de las implementaciones (arrays y listas) y veremos por qué el Montículo se considera particularmente eficiente.

### 18.2. Análisis de eficiencia

Cuando comparamos algoritmos, nos gustaría tener una forma de decir que uno es más rápido que otro, o que consume menos espacio, o que utiliza menos algún otro recurso. Es difícil responder a estas preguntas con detalle, porque el tiempo y el espacio usados por un algoritmo dependen de la implementación del algoritmo, el problema particular que se está resolviendo, y el hardware sobre el que se ejecuta el programa.

El objetivo de este apartado es establecer una manera de hablar sobre eficiencia que sea independiente de todas esas cosas, y que sólo dependa del algoritmo en sí mismo. Para empezar, nos centraremos en el tiempo de ejecución; después hablaremos de otros recursos.

Nuestras decisiones se guían por una serie de restricciones:

1. Primero, la eficiencia de un algoritmo depende del hardware sobre el que se ejecuta, por lo que normalmente no hablaremos de tiempo de ejecución en términos absolutos como segundos o minutos. En su lugar, normalmente contaremos el número de operaciones abstractas que ejecuta el algoritmo.
2. Segundo, la eficiencia a menudo depende del problema particular que intentamos resolver —algunos problemas son más fáciles que otros—. Para comparar algoritmos, normalmente nos centraremos en el peor caso o en un caso medio (más común).
3. Tercero, la eficiencia depende del tamaño del problema (normalmente, pero no siempre, el número de elementos de una colección). Reflejaremos esta dependencia de forma explícita expresando el tiempo de ejecución como una función del tamaño del problema.
4. Finalmente, la eficiencia depende de detalles de implementación como el tiempo de creación de objetos y el retardo en las llamadas a métodos. Normalmente ignoraremos esos detalles porque no afectan a la manera en cómo se incrementa el número de operaciones abstractas a medida que se incrementa el tamaño del problema.

Para hacer este proceso más concreto, consideremos dos algoritmos que hemos visto ya para ordenar un array de enteros. El primero la **ordenación por selección**, que vimos en el apartado 12.2. Este es el pseudocódigo que usamos allí:

```
ordenar_seleccion(array) is
  local
    i: INTEGER
  do
    from
      i := array.lower
    until
      i > array.upper
    loop
      -- calcular el menor elemento a la derecha de 'i'
      -- intercambiarlo con el i-ésimo elemento
      i := i + 1
    end
  end
```

Para realizar las operaciones especificadas en el pseudocódigo, escribimos métodos auxiliares llamados `calcular_menor` e `intercambiar`. En pseudocódigo, `calcular_menor` quedaría así

```
-- calcula el índice del menor elemento
-- entre 'i' y el final del array

calcular_menor(array, i) is
  local
    j: INTEGER
  do
    -- 'menor' contiene el índice del
    -- menor elemento hasta
```

```

    menor := i
    from
      j := i + 1
    until
      j > array.upper
    loop
      -- compara el j-ésimo elemento con el
      -- menor elemento hasta ahora
      -- si el j-ésimo elemento es menor, reemplazarlo por j
      j := j + 1
    end
    Result := menor
  end
end

```

E intercambiar quedaría así:

```

intercambiar(i, j) is
  do
    -- almacena una referencia a la i-ésima carta en 'temp'
    -- hace que el elemento i-ésimo del array apunte a la j-ésima carta
    -- hace que el j-ésimo elemento del array apunte a 'temp'
  end
end

```

Para analizar la eficiencia de este algoritmo, el primer paso es decidir qué operaciones se van a contar. Obviamente, el programa hace un montón de cosas: incrementa  $i$ , lo compara con la longitud de la baraja, busca el elemento más grande del array, etc. No está claro qué es lo que hay que contar.

Parece que una buena elección sería el número de veces que comparamos dos elementos. Otras muchas elecciones podrían proporcionarnos el mismo resultado, pero así resulta fácil de hacer y veremos que eso nos permitirá luego comparar más fácilmente este algoritmo con otros algoritmos de ordenación.

El siguiente paso es definir el “tamaño del problema”. En este caso parece lógico escoger el tamaño del array, al que podemos llamar  $n$ .

Finalmente, podemos querer obtener una expresión que nos diga cuántas operaciones abstractas (específicamente, comparaciones) tenemos que hacer, como una función de  $n$ .

Comenzaremos analizando los métodos ayudantes. `intercambiar` copia varias referencias, pero no realiza ninguna comparación, así que ignoraremos el tiempo consumido realizando intercambios. `calcular_menor` comienza en  $i$  y recorre el array, comparando cada elemento con `menor`. El número de elementos que se visitan es  $n - i$ , por lo que el número total de comparaciones es  $n - i - 1$ .

Ahora consideramos cuántas veces se llama a `calcular_menor` y cuál es el valor de  $i$  en cada ocasión. La última vez que se le invoca,  $i$  es  $n - 2$  por lo que el número de comparaciones es 1. La iteración anterior realiza 2 comparaciones, y así. Durante la primera iteración,  $i$  es 0 y el número de comparaciones es  $n - 1$ .

Por lo tanto, el número total de comparaciones es  $1 + 2 + \dots + n - 1$ . Esta suma es igual a  $n^2/2 - n/2$ . Para describir este algoritmo, podemos ignorar el término de orden inferior ( $n/2$ ) y decir que la cantidad total de trabajo es proporcional a  $n^2$ . Como el término de orden principal es cuadrático, también podemos decir que este algoritmo es de **tiempo cuadrático**.

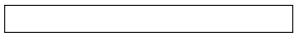



### 18.3. Análisis de la ordenación por mezcla

En el apartado 12.5 yo afirmaba que la ordenación por mezcla consume un tiempo que es proporcional a  $n \log n$ , pero no expliqué cómo o por qué. Ahora voy a hacerlo.

Nuevamente, empezamos echando un vistazo al pseudocódigo de este algoritmo. Para la ordenación por mezcla, es

```
ordenar_mezcla(array) is
do
    -- calcular el punto medio del array
    -- dividir el array en dos mitades
    -- ordenar cada mitad recursivamente
    -- mezclar las dos mitades y devolver el resultado
end
```

En cada nivel de recursión, dividimos en array por la mitad, hacemos dos llamadas recursivas, y después mezclamos las mitades. Gráficamente, el proceso se parece a ésto:

	arrays	elementos por array	mezclas	comparaciones por mezcla	trabajo total
	1	$n$	1	$n-1$	$\sim n$
	2	$n/2$	2	$n/2-1$	$\sim n$
⋮	⋮	⋮	⋮	⋮	⋮
	$n/2$	2	$n/2$	$2-1$	$\sim n$
	$n$	1	0	0	

Cada línea del diagrama es un nivel de recursión. Arriba del todo, un sólo array se divide en dos mitades. Abajo del todo,  $n$  arrays (con un sólo elemento cada uno) se mezclan en  $n/2$  arrays (con 2 elementos cada uno).

Las primeras dos columnas de la tabla muestran el número de arrays en cada nivel y el número de elementos de cada array. La tercera columna muestra el número de mezclas que tienen lugar en cada nivel de recursión. La siguiente columna es la que nos hace pensar más: muestra el número de comparaciones que realiza cada mezcla.

Si miras el pseudocódigo (o tu implementación) de la mezcla, puedes convencerte a ti mismo de que en el peor caso realiza  $m - 1$  comparaciones, donde  $m$  es el número total de elementos que están siendo mezclados.

El siguiente paso es multiplicar el número de mezclas en cada nivel por la cantidad de trabajo (comparaciones) por mezcla. El resultado es el trabajo total realizado en cada nivel. En este momento podemos sacar provecho de un pequeño truco. Sabemos que al final sólo estamos interesados en el término de mayor orden del resultado, por lo que podemos ignorar el término  $-1$  de las comparaciones por mezcla. Si hacemos eso, entonces el trabajo total realizado en cada nivel es simplemente  $n$ .

Después tenemos que conocer el número de niveles como una función de  $n$ . Bueno, comenzamos con un array de  $n$  elementos y lo vamos dividiendo por la mitad hasta que queda 1. Eso es lo mismo que comenzar con 1 e ir multiplicando por 2 hasta que obtenemos  $n$ . En otras palabras, queremos saber cuántas veces tenemos que multiplicar el número 2 por sí mismo para obtener  $n$ . La respuesta es que el número de niveles,  $l$ , es el logaritmo en base 2 de  $n$ .

Finalmente, multiplicamos la cantidad de trabajo realizado en cada nivel,  $n$ , por el número de niveles,  $\log_2 n$  para obtener  $n \log_2 n$ , como prometí. No hay un nombre apropiado para esta forma funcional; la mayoría de las veces la gente dice simplemente “ene logaritmo ene”.

Puede que no resulte obvio al principio el que  $n \log_2 n$  es mejor que  $n^2$ , pero para valores grandes de  $n$ , lo es.

*Como ejercicio, escribe un programa que imprima  $n \log_2 n$  y  $n^2$  para un rango de valores de  $n$ .*

## 18.4. Sobrecarga

El análisis de la eficiencia conlleva mucha manga ancha. Primero ignoramos muchas de las operaciones que realiza el programa y contamos sólo las comparaciones. Después decidimos considerar únicamente la eficiencia en el peor caso. Durante el análisis nos hemos tomado la libertad de redondear unas cuantas cosas, y cuando terminamos, descartamos los términos de menor orden.

Cuando interpretamos los resultados de este análisis, tenemos que tener en mente toda esta manga ancha. Como la ordenación por mezcla es  $n \log_2 n$ , lo consideramos un mejor algoritmo que la ordenación por selección, pero eso no significa que la ordenación por mezcla sea *siempre* más rápido. Tan sólo significa que si ordenamos arrays más y más grandes cada vez, la ordenación por mezcla ganará.

Cuánto tarde en realidad depende de los detalles de la implementación, incluyendo el trabajo adicional que realiza cada algoritmo, además de las comparaciones que hemos contado. Este trabajo extra a menudo se llama **sobrecarga**. No afecta al análisis de la eficiencia, pero sí que afecta al tiempo de ejecución del algoritmo.

Por ejemplo, nuestra implementación de ordenación por mezcla en realidad crea subarrays antes de hacer las llamadas recursivas y luego deja que sean eliminados por el recolector de basura después de haber sido mezclados. Observando de nuevo el diagrama de la ordenación por mezcla, podemos ver que la cantidad total de espacio que consume es proporcional a  $n \log_2 n$ , y el número total de objetos que se crean está en torno a  $2n$ . Toda esta creación lleva su tiempo.

Más aún, a menudo se cumple que una mala implementación de un buen algoritmo es mejor que una buena implementación de un mal algoritmo. La razón es que para valores grandes de  $n$  el algoritmo bueno es mejor y para valores pequeños de  $n$  no importa porque ambos algoritmos son lo bastante buenos.

*Como ejercicio, escribe un programa que imprima valores de  $1000n \log_2 n$  y  $n^2$  para un rango de valores de  $n$ . ¿Para qué valores de  $n$  llegan a ser iguales?*

## 18.5. Implementaciones de Colas con prioridad

En el capítulo 16 vimos una implementación de una Cola con prioridad basada en un array. Los elementos del array no están ordenados, por lo que es fácil añadir un nuevo elemento (al final), pero resulta más difícil eliminar un elemento, porque tenemos que buscar aquél que tenga la prioridad más alta.

Una alternativa es una implementación basada en una lista ordenada. En este caso cuando insertamos un nuevo elemento recorremos la lista y colocamos el nuevo elemento en la posición correcta. Esta implementación aprovecha una propiedad de las listas, por la que resulta fácil insertar un nuevo nodo en medio. De manera similar, eliminar el elemento con mayor prioridad es fácil, pues lo mantenemos al comienzo de la lista.

El análisis de la eficiencia de estas operaciones es inmediato. Añadir un nuevo elemento al final del array o eliminar un elemento del comienzo del array conlleva la misma cantidad de tiempo independientemente del número de elementos. Por lo tanto ambas operaciones son de tiempo constante.

Cada vez que recorremos una lista, realizando una operación de tiempo constante sobre cada elemento, el tiempo de ejecución es proporcional al número de elementos. Por tanto, eliminar algo del array y añadir algo a la lista supone en ambos casos un tiempo lineal.

Así que, ¿cuánto tiempo supone insertar y luego eliminar  $n$  elementos de una Cola con prioridad? Para la implementación basada en array, las  $n$  inserciones suponen un tiempo proporcional a  $n$ , pero las eliminaciones llevan más tiempo. La primera eliminación tiene que recorrer los  $n$

elementos; la segunda tiene que recorrer  $n - 1$ , y así, hasta la última eliminación, que sólo tiene que mirar en 1 elemento. Por tanto, el tiempo total es  $1 + 2 + \dots + n$ , que es (todavía)  $n^2/2 - n/2$ . Por lo que el total de inserciones y eliminaciones es la suma de una función lineal y una función cuadrática. El término de mayor orden es cuadrático.

El análisis de la implementación basada en lista es similar. La primera inserción no requiere ningún recorrido, pero después de eso tenemos que recorrer al menos parte de la lista cada vez que insertemos un nuevo elemento. En general no sabemos cuántos elementos de la lista se tendrán que recorrer, porque depende de los datos y en qué orden han sido insertados, pero podemos suponer que por término medio tenemos que recorrer la mitad de la lista. Por desgracia, incluso recorrer media lista es todavía una operación lineal.

Así que, de nuevo, insertar y eliminar  $n$  elementos supone un tiempo proporcional a  $n^2$ . Por tanto, basado en este análisis no podemos decir qué implementación es mejor; tanto el array como la lista proporcionan tiempos de ejecución cuadráticos.

Si implementamos una Cola con prioridad usando un montículo, podremos realizar las inserciones y las eliminaciones en un tiempo proporcional a  $\log n$ . Por tanto el tiempo total para  $n$  elementos es  $n \log n$ , lo que resulta mejor que  $n^2$ . Por esto es por lo que dije, al principio del capítulo, que un montículo es una implementación particularmente eficiente de una Cola con prioridad.

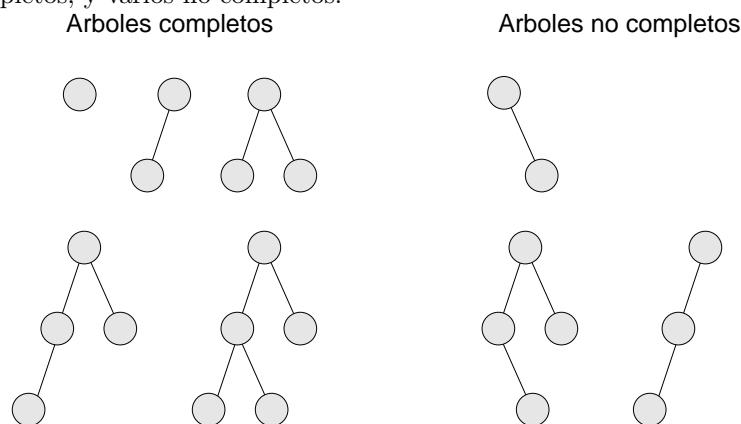
## 18.6. Definición de Montículo

Un montículo es un caso especial de árbol. Tiene dos propiedades que no resultan ciertas en general para otros árboles:

**completitud:** El árbol está completo, lo que significa que los nodos se añaden de arriba abajo, y de izquierda a derecha, sin dejar espacios.

**montículo:** El elemento con mayor prioridad está en la parte superior del árbol, y lo mismo se cumple para cada subárbol. Es decir, un nodo siempre tiene más prioridad que cualquiera de sus hijos.

Ambas propiedades merecen una pequeña explicación. Esta figura muestra varios árboles que se consideran completos, y varios no completos:



Un árbol vacío también se considera completo. Podemos definir la completitud de forma más rigurosa comparando la altura de los subárboles. Recuerda que la **altura** de un árbol es el número de niveles.

Comenzando por la raíz, si el árbol está completo, entonces la altura del subárbol izquierdo y la altura del subárbol derecho deben ser iguales, o el subárbol izquierdo tener una altura que

sea uno más que la altura del subárbol derecho. En cualquier otro caso, el árbol no puede ser completo.

Más aún, si el árbol está completo, entonces la relación de altura entre los subárboles tiene que cumplirse para cada nodo del árbol.

Es natural escribir esas reglas como un método recursivo:

```
class RAIZ
  -- ...
  feature
    -- ...
    es_completo(ar: ARBOL[G]): BOOLEAN is
      local
        altura_izq, altura_der, dif: INTEGER
      do
        -- el árbol vacío es completo
        if ar = Void then
          Result := True
        else
          altura_izq := altura(ar.izq)
          altura_der := altura(ar.der)
          dif := altura_izq - altura_der
          -- comprueba el nodo raíz
          if dif < 0 or dif > 1 then
            Result := False
          - comprueba los hijos
          elseif not es_completo(ar.izq) then
            Result := False
          else
            Result := es_completo(ar.der)
          end
        end
      end
    end
  end
end
```

*Para este ejemplo he utilizado la implementación enlazada de un árbol. Como ejercicio, escribe el mismo método para la implementación basada en array. Además, como ejercicio, escribe el método `altura`. La altura del árbol `Void` es 0 y la altura de un nodo hoja es 1.*

La **propiedad de montículo** también es recursiva. Para que un árbol se convierta en un montículo, el valor más grande del árbol tiene que estar en la raíz, y lo mismo debe cumplirse para cada subárbol.

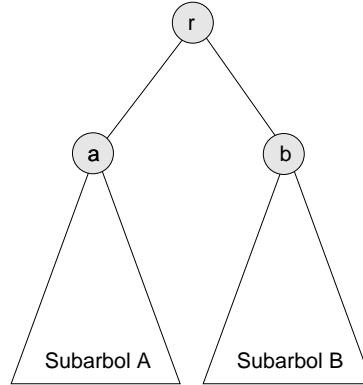
*Como otro ejercicio, escribe un método que compruebe si un árbol tiene la propiedad de montículo.*

## 18.7. Eliminar de un montículo

Puede resultar extraño que vayamos a eliminar cosas del montículo antes de insertar ninguna, pero creo que la eliminación es más fácil de explicar.

A primera vista, podemos pensar que eliminar un elemento del montículo es una operación constante en el tiempo, ya que el elemento de mayor prioridad siempre está en la raíz. El problema es que una vez hemos eliminado el nodo raíz, nos quedamos con algo que no es, ni de lejos, un montículo. Antes de poder devolver el resultado, tenemos que restituir la propiedad de montículo.

La situación se muestra en la siguiente figura:



El nodo raíz tiene prioridad  $r$  y dos subárboles, A y B. El valor de la raíz del subárbol A es  $a$  y el valor de la raíz del subárbol B es  $b$ .

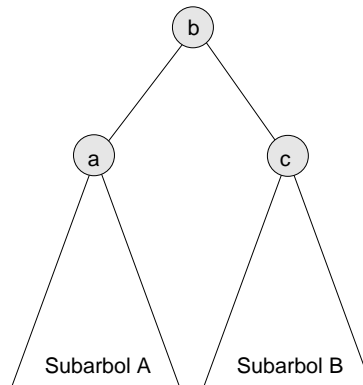
Suponemos que antes de eliminar  $r$  del árbol, ese árbol es un montículo. Esto implica que  $r$  es el valor más grande del montículo y que  $a$  y  $b$  son los valores más grandes en sus respectivos subárboles.

Una vez que hayamos eliminado  $r$ , tenemos que hacer que el árbol resultante sea de nuevo un montículo. En otras palabras, necesitamos asegurarnos que tendrá las propiedades de completitud y montículo.

La mejor manera de asegurar la completitud es eliminar el nodo situado más abajo y más a la derecha, un nodo al que llamaremos  $c$ , y colocar su valor en la raíz. En la implementación de un árbol general, tendríamos que recorrer el árbol para encontrar ese nodo, pero en la implementación basada en array, podemos encontrarlo en tiempo constante porque es siempre el último elemento (no vacío) del array.

Por supuesto, existe la posibilidad de que el último valor no sea el más grande, por lo que colocarlo en la raíz rompería la propiedad de montículo. Afortunadamente, es fácil restituirla. Sabemos que el valor más grande del montículo es  $a$  o  $b$ . Por tanto, podemos seleccionar el que sea más grande e intercambiarlo con el valor de la raíz.

De manera arbitraria, digamos que  $b$  es más grande. Como sabemos que es el mayor valor que queda en el montículo, podemos ponerlo en la raíz y colocar  $c$  en la cima del subárbol B. Ahora la situación queda así:



De nuevo,  $c$  es el valor que copiamos de la última entrada en el array y  $b$  es el mayor valor que queda en el montículo. Como no hemos cambiado para nada el subárbol A, sabemos que éste



todavía es un montículo. El único problema es que no sabemos si el subárbol B es un montículo, porque acabamos de poner un valor (posiblemente pequeño) en su raíz.

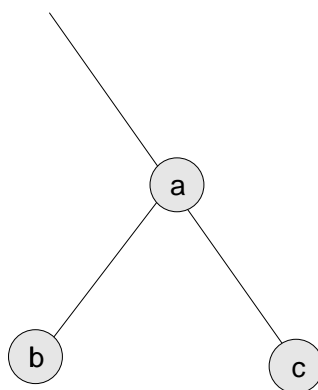
¿No sería estupendo tener un método que pudiera restituir la propiedad de montículo en el subárbol B? Espera... ¡lo tenemos!

## 18.8. insertar en un montículo

Insertar un nuevo elemento en el montículo es una operación similar, con la diferencia de que, en lugar de hundirlo desde arriba, lo hacemos flotar desde abajo.

De nuevo, para garantizar la completitud, añadimos el nuevo elemento en la posición más abajo y más a la derecha del árbol, que es el siguiente espacio disponible en el array.

Después de restituir la propiedad de montículo, comparamos el nuevo valor con sus vecinos. La situación se parece a ésta:



El nuevo valor es *c*. Podemos restituir la propiedad de montículo de este subárbol comparando *c* con *a*. Si *c* es más pequeño, entonces se satisface la propiedad de montículo. Si *c* es más grande, entonces intercambiamos *c* y *a*. El intercambio satisface la propiedad de montículo porque sabemos que *c* también debe ser más grande que *b*, porque  $c > a$  y  $a > b$ .

Ahora que el subárbol ha restituido la propiedad de montículo, podemos seguir hacia arriba por el árbol hasta alcanzar la raíz.

## 18.9. Eficiencia de los montículos

Insertar y eliminar son operaciones en tiempo constante, pero después de realizarlas tenemos que restituir la propiedad de montículo. En un caso empezamos en la raíz y vamos bajando, comparando elementos y después, recursivamente, restituyendo la propiedad de montículo en uno de los subárboles. En el otro caso empezamos en una hoja y vamos subiendo, de nuevo comparando elementos en cada nivel del árbol.

Como es habitual, existen varias operaciones que podemos contar, como las comparaciones y los intercambios. Cualquier elección podría valer; la verdadera cuestión es el número de niveles que tenemos que examinar del árbol y cuánto trabajo tenemos que hacer en cada nivel. En ambos casos vamos examinando niveles del árbol hasta que restituimos la propiedad de montículo, lo que significa que podemos visitar sólo uno, o en el peor caso tenemos que visitar todos ellos. Consideremos el peor caso.

En cada nivel, realizamos únicamente operaciones de tiempo constante como comparaciones e intercambios. Por lo que la cantidad total de trabajo es proporcional al número de niveles del árbol, o sea, la altura.

Así que podemos decir que esas operaciones son lineales respecto a la altura del árbol, pero el “tamaño del problema” en el que estamos interesados no es la altura, sino el número de elementos del montículo.

En función de  $n$ , la altura del árbol es  $\log_2 n$ . Esto no es cierto para todos los árboles, pero se cumple en los árboles completos. Para ver por qué, piensa en el número de nodos que hay en cada nivel del árbol. El primer nivel contiene 1, el segundo contiene 2, el tercero contiene 4, y así. El nivel  $i$ -ésimo contiene  $2^i$  nodos, y el número total en todos los niveles hasta el nivel  $i$  es  $2^i - 1$ . En otras palabras,  $2^h = n$ , lo que supone que  $h = \log_2 n$ .

Por tanto, tanto la inserción como la eliminación consumen un tiempo **logarítmico**. Insertar y eliminar  $n$  elementos supone un tiempo proporcional a  $n \log_2 n$ .

## 18.10. Heapsort

El resultado del apartado anterior sugiere otro algoritmo más de ordenación. Dados  $n$  elementos, los insertamos en un Montículo y luego los eliminamos. Debido a la semántica del Montículo, saldrán en orden. Ya hemos visto que este algoritmo, llamado **heapsort**<sup>1</sup>, consume un tiempo proporcional a  $n \log_2 n$ , lo que es mejor que la ordenación por selección, e igual que la ordenación por mezcla.

A medida que el valor de  $n$  crece, esperamos que heapsort sea más rápido que la ordenación por selección, pero el análisis de la eficiencia no nos proporciona una manera de saber si será más rápido que la ordenación por mezcla. Podemos decir que los dos algoritmos tienen el mismo **orden de crecimiento** porque crecen con la misma forma funcional. Otra manera de decir lo mismo es que pertenecen a la misma **clase de complejidad**.

Las clases de complejidad a veces se escriben en “notación O mayúscula”. Por ejemplo,  $\mathcal{O}(n^2)$ , que se lee “O de ene cuadrado” es el conjunto de todas las funciones que crecen no más deprisa que  $n^2$  para valores grandes de  $n$ . Decir que un algoritmo es  $\mathcal{O}(n^2)$  es lo mismo que decir que es cuadrático. Las otras clases de complejidad que hemos visto, en orden decreciente de eficiencia, son:

$\mathcal{O}(1)$	tiempo constante
$\mathcal{O}(\log n)$	logarítmico
$\mathcal{O}(n)$	lineal
$\mathcal{O}(n \log n)$	“ene logaritmo ene”
$\mathcal{O}(n^2)$	cuadrático
$\mathcal{O}(2^n)$	exponencial

Hasta ahora no hemos visto ningún algoritmo que sea **exponencial**. Para valores grandes de  $n$ , esos algoritmos dejan de ser prácticos rápidamente. No obstante, la frase “crecimiento exponencial” aparece frecuentemente incluso en el lenguaje no técnico. Muchas veces se utiliza de forma incorrecta, por lo que quisiera exponer su significado técnico.

La gente a menudo usa “exponencial” para describir toda curva que crece y se acelera (esto es, que tiene una pendiente y curvatura positiva). Por supuesto, hay muchas otras curvas que entran en esta descripción, incluidas las funciones cuadráticas (y los polinomios de orden superior) e incluso funciones como  $n \log n$ . Muchas de esas curvas no tienen el (a menudo indeseable) explosivo crecimiento de las exponenciales.

Como ejercicio, compara el comportamiento de  $1000n^2$  y  $2^n$  a medida que crece el valor de  $n$ .

<sup>1</sup>Que puede traducirse como *ordenación por montículo*.

## 18.11. Glosario

**ordenación por selección:** El algoritmo de ordenación del apartado 12.2.

**ordenación por mezcla:** Un mejor algoritmo de ordenación que vimos en el apartado 12.5.

**heapsort:** Otro algoritmo más de ordenación.

**clase de complejidad:** Un conjunto de algoritmos cuya eficiencia (normalmente tiempo de ejecución) tiene el mismo orden de crecimiento.

**orden de crecimiento:** Un conjunto de funciones con el mismo término de orden superior, y por tanto el mismo comportamiento cualitativo para valores grandes de  $n$ .

**sobrecarga:** Tiempo o recursos adicionales consumidos durante la realización de operaciones de programación distintas a las operaciones abstractas consideradas en el análisis de la eficiencia.

## Capítulo 19

# Tablas

### 19.1. Arrays y Tablas

Los arrays son habitualmente una útil estructura de datos, pero sufren dos importantes limitaciones:

- El tamaño del array no depende del número de elementos que contiene. Si el array es demasiado grande, malgasta espacio. Si es demasiado pequeño, puede provocar un error, u obligarnos a redimensionarlo.
- Aunque el array puede contener cualquier tipo de elemento, los índices del array deben ser enteros. No podemos, por ejemplo, usar una cadena para especificar un elemento del array.

Aquí es donde entra el TAD Tabla. La Tabla es una generalización del `ARRAY` que puede usar cualquier tipo con índice. Esos índices generalizados se llaman **claves**.

Al igual que utilizas un índice para acceder a un valor en un array, puedes usar una clave para acceder a un valor en una Tabla. Por tanto, cada clave está asociada a un valor, lo que hace que a las tablas a veces se las denomine **arrays asociativos**.

Un ejemplo típico de tabla es un diccionario, que es una tabla que asocia palabras (las claves) con sus definiciones (los valores). Debido a este ejemplo a las Tablas a menudo se las llama Diccionarios. Además, a la asociación de una clave particular con un valor particular se le llama **entrada**.

### 19.2. El TAD Tabla

Como los otros TADs que hemos visto, las Tablas se definen por el conjunto de operaciones que soportan:

**make:** Crea una nueva tabla vacía.

**meter:** Crea una entrada que asocia un valor con una clave.

**obtener:** Dada una cierta clave, busca su correspondiente valor.

**contiene\_clave:** Devuelve `True` si existe una entrada en la Tabla con la clave dada.

**claves:** Devuelve una colección que contiene todas las claves de la Tabla.

### 19.3. La clase DS\_HASH\_TABLE

Eiffel proporciona, a través de la librería GOBO, una implementación del TAD Tabla llamada DS\_HASH\_TABLE.

Para mostrar el uso de DS\_HASH\_TABLE escribiremos un pequeño programa que recorra una cadena y cuente el número de veces que aparece cada palabra.

Crearemos una nueva clase llamada CUENTA\_PALABRAS que construirá la Tabla y después imprimirá su contenido. Naturalmente, cada objeto CUENTA\_PALABRAS contendrá un DS\_HASH\_TABLE:

```
class CUENTA_PALABRAS

  creation make

  feature {NONE}
    ht: DS_HASH_TABLE[INTEGER, STRING]

  feature
    make is
    do
      create ht.make_default
    end
  end

end
```

Como podrás observar, el atributo `ht` no es accesible desde fuera de la clase. Los únicos métodos públicos de CUENTA\_PALABRAS, además del constructor, son `procesar_linea`, que recibe una cadena y añade sus palabras a la Tabla, e `imprimir`, que imprime los resultados al final.

`procesar_linea` rompe la cadena en palabras (para lo cual podemos usar el método predefinido `split`), y pasa cada palabra a `procesar_palabra`.

```
class CUENTA_PALABRAS
  -- ...
  feature
    -- ...
    procesar_linea(s: STRING) is
      local
        a: ARRAY[STRING]
        i: INTEGER
      do
        a := s.split
        from
          i := a.lower
        until
          i > a.upper
        loop
          palabra := a @ i
          procesar_palabra(palabra.as_lower)
          i := i + 1
        end
      end
    end
  end
end
```

Con `palabra.as_lower` lo que hacemos es obtener la misma palabra pero con todos sus caracteres en minúscula. Esto se hace para que, por ejemplo, las palabras “Hola” y “HOLA” sean contadas como una sola. Pero lo más interesante está en `procesar_palabra`.

```

class CUENTA_PALABRAS
  -- ...
feature
  -- ...
  procesar_palabra(palabra: STRING) is
    local
      i, j: INTEGER
    do
      if ht.has(palabra) then
        i := ht.item(palabra)
        j := i + 1
        ht.put(j, palabra)
      else
        ht.put(1, word)
      end
    end
  end
end

```

Si la palabra ya estaba en la tabla, obtenemos su contador, lo incrementamos, y le asignamos el nuevo valor. En caso contrario, simplemente insertamos una nueva entrada en la tabla con el contador puesto a 1.

Para imprimir las entradas de la tabla, tenemos que ser capaces de recorrer las claves de la tabla. Afortunadamente, la implementación `DS_HASH_TABLE` proporciona métodos que nos permiten recorrer la tabla mediante un *cursor*, que es algo muy similar a los iteradores que vimos en el apartado 17.12. Así se usa el cursor interno para imprimir el contenido de la Tabla:

```

class CUENTA_PALABRAS
  -- ...
feature
  -- ...
  imprimir is
    local
      clave: STRING
      valor: INTEGER
    do
      from
        ht.start
      until
        ht.off
      loop
        clave := ht.key_for_iteration
        valor := ht.item_for_iteration
        print("{ " + clave + ", " + valor.to_string + " }%N")
        ht.forth
      end
    end
  end
end

```

`start` sitúa el cursor en la primera entrada de la tabla. `key_for_iteration` e `item_for_iteration` devuelven, respectivamente, la clave y el valor de la entrada en la que está situado el cursor. `off` devuelve `True` si el cursor ha pasado el final de la tabla. Y `forth` avanza el cursor una entrada hacia delante. Como ves, se parece bastante a la forma de trabajar con iteradores.

Finalmente, para contar las palabras de una cadena:

```

class RAIZ

creation make

feature
  make is
    local
      cp: CUENTA_PALABRAS
    do
      create cp.make
      cp.procesar_linea("da doo ron ron ron, da doo ron ron")
      cp.imprimir
    end
end
end

```

La salida es

```

{ ron, 5 }
{ doo, 2 }
{ da, 2 }

```

Los elementos devueltos por el cursor no se encuentran en un determinado orden. La única garantía es que aparecen todas las entradas de la tabla.

## 19.4. Implementación basada en array

Una manera sencilla de implementar el TAD Tabla es usar un **ARRAY** de entradas, donde cada entrada es un objeto que contiene una clave y un valor. A estos objetos les llamaremos **pares clave-valor**.

Una definición de clase para un **PAR\_CLAVE\_VALOR** podría parecerse a esto:

```

class PAR_CLAVE_VALOR[K, V]

creation make

feature
  clave: K
  valor: V

  make(cl: K; va: V) is
    do
      clave := cl
      valor := va
    end

  to_string: STRING is
    do
      Result := "{ " + clave.to_string + ", " + valor.to_string + " }"
    end
end
end

```

Y entonces la implementación de la Tabla sería algo así:

```

class TABLA[K, V]

creation make

feature
  vector: ARRAY[PAR_CLAVE_VALOR[K, V]]

  make is
    do
      create vector.make(1, 10)
    end
end

```

Para almacenar una nueva entrada en la tabla, simplemente añadimos un nuevo `PAR_CLAVE_VALOR` en el array:

```

class TABLA[K, V]
  -- ...
feature
  -- ...
  meter(cl: K; va: V) is
    local
      par: PAR_CLAVE_VALOR[K, V]
    do
      create par.make(cl, va)
      vector.add_last(par)
    end
end

```

Después para buscar una clave en la Tabla tenemos que recorrer el array y encontrar un `PAR_CLAVE_VALOR` cuya clave coincida con la clave buscada:

```

class TABLA[K, V]
  -- ...
feature
  -- ...
  obtener(cl: K): V is
    local
      i: INTEGER
      par: PAR_CLAVE_VALOR[K, V]
      encontrado: BOOLEAN
    do
      from
        i := vector.lower
        encontrado := False
      until
        i > vector.upper or encontrado
      loop
        par := vector @ i
        if cl.is_equal(par.clave) then
          Result := par.valor
        end
        i := i + 1
      end
    end
  end

```



```

        end
        if not encontrado then
            Result := Void
        end
    end
end
end

```

Cuando comparamos claves, usamos la igualdad profunda (el método `is_equal`) en lugar de la igualdad superficial (el operador `=`). Esto permite que la clase a la que pertenecen las claves pueda especificar la definición de igualdad. En nuestro ejemplo, las claves son cadenas, por lo que utilizará el método `is_equal` predefinido en la clase `STRING`.

Como `is_equal` es un método de objeto, esta implementación de `obtener` no funciona si la clave `cl` es `Void`. Podríamos manejar `Void` como un caso especial, o podríamos incluir una precondition que obligue a que toda clave sea distinta de `Void`.

Los únicos métodos que no hemos implementado son `contiene_clave` y `claves`. El método `contiene_clave` es casi idéntico a `obtener` excepto en que devuelve `True` o `False` en lugar de una referencia al objeto, o `Void`.

*Como ejercicio, implementa `claves` construyendo un array de claves y devolviendo los elementos del vector.*

## 19.5. La clase diferida `COLLECTION`

Eiffel define una clase diferida llamada `COLLECTION` que especifica el conjunto de operaciones que una clase tiene que implementar para poder ser considerada (de manera muy abstracta) una colección de elementos. Esto no significa, por supuesto, que toda clase que herede de `COLLECTION` y redefina sus métodos `deferred` tenga que ser una lista enlazada.

No es sorprendente que la clase `LINKED_LIST` sea un miembro de la clase diferida `COLLECTION`. Curiosamente, también lo es la clase `ARRAY`.

Los métodos de la definición de `COLLECTION` incluyen `put`, `item` y `get_new_iterator`. De hecho, todos los métodos de la clase `ARRAY` que hemos usado para implementar la Tabla están definidos en la clase diferida `COLLECTION`.

Esto significa que en lugar de un `ARRAY`, podríamos haber usado cualquier clase perteneciente al conjunto `COLLECTION`. En `tabla.e` podemos substituir `ARRAY` por `LINKED_LIST`, ¡y el programa seguiría funcionando!

Este tipo de genericidad puede ser útil para sintonizar la eficiencia de un programa. Puedes escribir el programa en términos de una clase diferida como `COLLECTION` y luego probar el programa con varias implementaciones de `COLLECTION` diferentes para ver cuál proporciona las mejores prestaciones.

## 19.6. Implementación de tabla hash

La razón por la que se llama `DS_HASH_TABLE` a la implementación predefinida del TAD Tabla es que utiliza una implementación particularmente eficiente de Tabla llamada tabla hash.

Por supuesto, lo importante de definir un TAD es que nos permite usar una implementación sin conocer los detalles. Por tanto, probablemente sea algo inadecuado el hecho de que la gente que escribió la librería Eiffel denominara a esa clase según su implementación en lugar de su TAD, pero supongo que de todas las cosas inadecuadas que hicieron, esta es muy pequeña.

En todo caso, te puedes estar preguntando qué es una tabla hash, y por qué digo que es particularmente eficiente. Comenzaremos analizando la eficiencia de la implementación de `COLLECTION` que acabamos de hacer.

Mirando la implementación de `put`, vemos que hay dos casos. Si la clave no está ya en la tabla, entonces sólo tenemos que crear un nuevo par clave-valor y añadirlo a la colección. Ambas operaciones consumen un tiempo constante.

En el otro caso, tenemos que recorrer la colección para encontrar el par clave-valor existente. Esta es una operación de tiempo lineal. Por la misma razón, `obtener` y `contiene_claves` también son lineales.

Aunque las operaciones lineales a menudo son bastante buenas, podemos hacerlo mejor. ¡Existe una manera de implementar el TAD tabla de forma que tanto `insertar` como `obtener` son operaciones en tiempo constante!

La cuestión es darse cuenta de que recorrer una colección consume un tiempo proporcional a la longitud de la colección. Si podemos poner un límite superior a la longitud de la colección, entonces podemos poner un límite superior al tiempo del recorrido, y todo aquello que consume un tiempo cuyo límite superior está prefijado se considera de tiempo constante.

Pero, ¿cómo podemos limitar la longitud de las colecciones sin limitar el número de elementos de la tabla? Incrementando el número de colecciones. En lugar de tener una colección larga, tendremos muchas colecciones cortas.

Si sabemos en qué colección tenemos que buscar, podemos acotar la cantidad de búsqueda que tenemos que realizar.

## 19.7. Funciones hash

Y aquí es donde aparecen las funciones hash. Necesitamos conseguir que, simplemente mirando la clave, sepamos sin hacer búsquedas en qué colección estará. Supondremos que las colecciones están en un array de manera que podamos referirnos a ellas mediante un índice.

La solución consiste en realizar alguna correspondencia —casi cualquier correspondencia— entre las claves y los índices de las colecciones. Para cada posible clave tiene que haber un único índice, pero pueden existir varias claves que se correspondan con el mismo índice.

Por ejemplo, imagina un array de 8 colecciones y una tabla compuesta de claves que son `INTEGERs` y de valores que son `STRINGs`. Puede resultar tentador usar los enteros como índices, ya que tienen el tipo adecuado, pero entonces existirían un montón de enteros que no caerían entre 0 y 7, que son los únicos índices permitidos.

El operador módulo proporciona una manera simple (en términos de codificación) y eficiente (en términos de tiempo de ejecución) de hacer corresponder *todos* los enteros con el rango `[0, 7]`. La expresión

```
clave % 8
```

seguro que produce un valor en el rango de `-7` to `7` (inclusive). Si tomas el valor absoluto, obtendrás un índice permitido.

Para otros tipos, podemos jugar a juegos similares. Por ejemplo, para convertir un `CHARACTER` en un entero, podemos usar el método predefinido `code`, y para los `DOUBLEs` podemos usar `truncated_to_integer`.

Para los `STRINGs` podemos obtener el valor numérico de cada carácter y sumarlos, o en su lugar podemos usar una **suma desplazada**. Para calcular una suma desplazada, alternamos entre añadir nuevos valores al acumulador y desplazar el acumulador a la izquierda. Por “desplazar a la izquierda” me refiero a “multiplicar por una constante”.

Para ver cómo funciona esto, vamos a tomar la lista de números 1, 2, 3, 4, 5, 6 y a calcular su suma desplazada como sigue. Primero, inicializamos el acumulador a 0. Después,

1. Multiplicamos el acumulador por 10.
2. Añadimos el siguiente elemento de la lista al acumulador.
3. Repetimos hasta que se acabe la lista.

*Como ejercicio, escribe un método que calcule la suma desplazada de los valores numéricos de los caracteres de una cadena usando un múltiplo de 32.*

Para cada posible tipo, podemos tener una función que reciba valores de ese tipo y genere el valor entero que le corresponda. Esas funciones se denominan **funciones hash**. El valor entero correspondiente a cada objeto se denomina su **código hash**.

Existe otra manera de generar un código hash para los objetos de Eiffel. Cada objeto Eiffel proporciona una característica llamada `hash_code` que devuelve el entero que se corresponde con ese objeto. Para los tipos predefinidos, `hash_code` está implementada de tal manera que si dos objetos contienen la misma información, tendrán el mismo código hash (como ocurre con la igualdad profunda). Cada clase tiene por tanto su propia función hash, y ésta se explica en la documentación de cada clase. Deberías echarles un vistazo.

Para los tipos definidos por el usuario, el implementador del tipo es el encargado de proporcionar una función hash apropiada. Si un objeto tiene definida la característica `hash_code`, se dice que dicho objeto es “hashable”. El conjunto de todos los tipos “hashables” posibles forman una clase diferida llamada `HASHABLE`, que proporciona la característica diferida `hash_code`. Toda clase “hashable” debe ser heredera de `HASHABLE`, y debe redefinir el método `hash_code` de acuerdo con sus necesidades.

Independientemente de cómo se genera el código hash, el último paso es usar el operador módulo y el valor absoluto para corresponder el código hash con el rango de los índices permitidos.

## 19.8. Redimensionar una tabla hash

Recapitemos. Una tabla hash consiste en un array de `COLLECTIONs`, donde cada `COLLECTION` contiene un pequeño número de pares clave-valor. Para añadir una nueva entrada a la tabla, calculamos el código hash de la nueva clave y añadimos la entrada en la correspondiente colección.

Para buscar una clave, calculamos de nuevo su código hash y buscamos en la correspondiente colección. Si la longitud de las colecciones está acotada, entonces el tiempo de búsqueda también estará acotado.

Pero, ¿cómo mantenemos cortas las colecciones? Bueno, un objetivo es mantenerlas tan equilibradas como sea posible, de manera que no se den colecciones muy largas al mismo tiempo que otras están vacías. Esto no es fácil de hacer perfectamente —depende de lo bien que elijamos la función hash— pero a veces podemos hacer un buen trabajo.

Incluso con un equilibrio perfecto, la longitud media de una colección crece linealmente con el número de entradas, y tenemos que parar eso.

La solución es seguir la pista del número medio de entradas por colección, lo que se denomina el **factor de carga**; si el factor de carga se hace muy grande, tenemos que redimensionar la tabla.

Para redimensionar, creamos una nueva tabla, a menudo con el doble de tamaño que la original, cogemos todas las entradas de la tabla vieja, recalculamos de nuevo sus códigos hash, y las colocamos en la nueva tabla. Normalmente podemos salir adelante usando la misma función hash; tan sólo usamos un valor diferente para el operador módulo.

## 19.9. Eficiencia de la redimensión

¿Cuánto se tarda en redimensionar la tabla? Claramente el tiempo es lineal con el número de entradas. Esto significa que *muchas* veces **insertar** consume un tiempo constante, pero de vez en cuando —cuando redimensionamos— consume un tiempo lineal.

Al principio eso suena mal. ¿No contradice mi afirmación de que podemos realizar insertar en tiempo constante? Bueno, francamente, sí. Pero con un poco de seducción, puedo arreglarlo.

Como algunas operaciones **insertar** consumen más tiempo que otras, tengamos en cuenta el tiempo *medio* de cada operación **insertar**. La media será  $c$ , el tiempo constante de un sencillo **insertar**, más el término adicional  $p$ , el porcentaje de veces que tenemos que redimensionar, multiplicado por  $kn$ , el coste de la redimensión.

$$t(n) = c + p \cdot kn \quad (19.9.1)$$

No sé cuánto valen  $c$  y  $k$ , pero puedo hacerme una idea de cuánto vale  $p$ . Imagina que acabamos de redimensionar la tabla hash haciendo que tenga el doble de tamaño. Si hay  $n$  entradas, entonces podemos añadir otras  $n$  entradas antes de tener que redimensionar de nuevo. Por tanto, el porcentaje de veces que tenemos que redimensionar es  $1/n$ .

Sustituyendo en la ecuación, tenemos

$$t(n) = c + 1/n \cdot kn = c + k \quad (19.9.2)$$

En otras palabras,  $t(n)$  es un tiempo constante!

## 19.10. Glosario

**tabla:** Un TAD que define operaciones sobre una colección de entradas.

**entrada:** Un elemento de una tabla, que contiene un par clave-valor.

**clave:** Un índice, de cualquier tipo, usado para buscar valores en una tabla.

**valor:** Un elemento, de cualquier tipo, almacenado en una tabla.

**diccionario:** Otra manera de llamar a las tablas.

**array asociativo:** Otra manera de llamar a los diccionarios.

**tabla hash:** Una implementación particularmente eficiente de una tabla.

**función hash:** Una función que hace corresponder valores de un cierto tipo con números enteros.

**código hash:** El valor entero que se corresponde con un valor dado.

**suma desplazada:** Una sencilla función hash usada a menudo por objetos compuestos como las cadenas.

**factor de carga:** El número de entradas de una tabla hash dividido por el número de colecciones en la tabla hash; es decir, el número medio de entradas por colección.

# Índice alfabético

- árbol, 177, 197
  - completo, 197
  - de expresiones, 180
  - implementación basada en array, 184
  - implementación enlazada, 178
  - recorrido, 179, 180
  - vacío, 179
- árbol binario, 177, 191
- árbol completo, 197
- árbol de expresiones, 180
- índice, 61, 68, 96, 105, 113, 203
- abstracción, 117, 118
- algoritmo, 93, 94
- alias, 77, 80, 110, 123
- alternativa, 31, 36
  - anidada, 33, 36
  - doble, 32
  - encadenada, 33, 36
- alternativa anidada, 33
- alternativa doble, 32
- alternativa encadenada, 33
- altura, 197
- ambigüedad, 5, 109
  - teorema fundamental, 144
- análisis
  - Cola con prioridad, 196
  - heapsort, 201
  - Montículo, 200
  - tabla hash, 209, 211
- análisis de algoritmos, 192, 194
- análisis de eficiencia, 161, 192
- análisis sintáctico, 5, 8
- andamiaje, 40, 47
- ANY, 137
- argumento, 21, 26, 30
- aritmética
  - complejos, 128
  - real, 19, 91
- array, 95, 105, 203
  - cambiar el tamaño, 158
  - copiar, 97
  - de CARTAs, 119
  - de objetos, 112
  - de STRING, 108
  - elemento, 96
  - recorrido, 101
- array asociativo, 203, 211
- asignación, 11, 17, 48
- atributo, 71, 80, 82, 119, 129
- búsqueda, 114
- búsqueda binaria, 114
- búsqueda lineal, 114
- baraja, 112, 117, 119
- barajar, 121, 124
- bucle, 49, 58, 97
  - anidado, 112
  - búsqueda, 114
  - contar, 64
  - cuerpo, 49
  - en lista, 143
  - infinito, 49, 58
  - inicialización, 49
- bucle infinito, 49, 58
- bucle y conteo, 101
- buffer
  - circular, 165
- buffer circular, 165, 175
- bug, 3
- buscar\_binaria, 115
- buscar\_carta, 114
- código hash, 210, 211
- cambiar\_cartas, 122
- carácter, 65
- características, 82
- carga, 139, 150, 177
- CARTA, 106
- CHARACTER, 59
- clase, 25, 30, 94
  - ARRAY, 189
  - CARTA, 106
  - COMPLEJO, 128
  - diferida, 169, 171, 182
  - DS\_HASH\_TABLE, 204
  - GOLFISTA, 172
  - HORA, 82
  - ITERATOR, 190

- LISTA\_ENLAZADA, 147
- NODO, 139
- nombre, 7
- padre, 137
- PUNTO, 70
- raíz, 81
- RECTANGULO, 73, 136
- STRING, 59, 66, 67
- TOKEN, 183
- clase ARRAY, 189
- clase de complejidad, 201, 202
- clase diferida, 169, 171, 175, 182, 190
  - COLLECTION, 208
  - COMPARABLE, 169
  - definición, 182
  - hacer efectiva, 172, 183
  - VISITABLE, 182
- clase diferida COLLECTION, 208
- clase ITERATOR, 190
- clase NODO, 139
- clase padre, 137
- clase raíz, 7, 81, 94
- clase TOKEN, 183
- clave, 203, 211
- cliente, 151, 159, 171, 182
- codificar, 106, 118
- Cola
  - implementación basada en buffer circular, 165
  - implementación basada en listas, 160
  - implementación enlazada, 163
- cola, 160, 175
- Cola con prioridad
  - implementación basada en array, 169
  - implementación basada en lista ordenada, 196
- TAD, 168
- cola con prioridad, 160, 175
- cola enlazada, 163, 175
- colección, 105, 141, 151, 160
- comentario, 7
- comillas dobles, 60
- comillas simples, 60
- COMPARABLE, 169, 172
- comparable, 111
- comparación
  - operador, 32
  - STRING, 67
- comparar\_carta, 110
- compare, 67
- compilar, 2, 8
- COMPLEJO, 128
- composición, 15, 17, 22, 41, 106, 112
- concatenar, 15, 17
- condición, 31
- constante en el tiempo, 162
- constructor, 81, 83, 94, 107, 119, 123, 129, 137
- contador, 53, 65, 68, 101
- contrato, 62, 63
- corrección, 117
- correspondencia, 209
- corresponder, 106
- count
  - STRING, 60
- create, 70, 85
- creation, 7, 120
- cuerpo
  - bucle, 49
  - método, 23, 183
- Current, 128, 138
- cursor, 205
- dar una mano, 124
- declaración, 11, 70
- decrementar, 65, 68
- deferred, 182
- definición de clase, 81
- definición recursiva, 198
- delimitador, 159
- depuración, 3, 8
- desarrollo de programas, 39, 58
  - incremental, 91
  - planificación, 91
- desarrollo incremental, 39, 91
- desbordamiento de pila, 36, 117
- determinístico, 99, 105
- diagrama
  - estado, 35, 45
  - implementación, 177, 192
  - pila, 35, 45
- diagrama de estado, 11, 70, 80, 95, 108, 112, 119
- diagrama de pila, 29
- dibujable, 136
- diccionario, 203, 211
- disciplina de encolado, 160, 175
- diseño orientado a objetos, 137
- Diseño por Contrato, 63
- distribución, 99
- división
  - real, 51
- división entera, 31
- documentación, 59, 62, 148
- double (real), 18
- Doyle, Arthur Conan, 4
- DS\_HASH\_TABLE, 204

- eficiencia, 124
- elemento, 96, 105
- encapsulación, 54, 55, 58, 65, 75, 113, 151, 155, 182, 191
- encolado por prioridad, 160
- encriptar, 106
- enlace, 150
- ensure, 63
- entrada, 203, 211
- envoltorio, 150, 161, 175
- error, 8
  - lógico, 4
  - tiempo de compilación, 3
  - tiempo de ejecución, 4, 61
- error en tiempo de compilación, 3
- error en tiempo de ejecución, 4, 61, 68, 78, 112
- error en tiempo de ejecución, 96
- error lógico, 4
- estadísticas, 99
- estado, 70, 80
- estilo de programación, 91, 127
- estructura anidada, 34, 42, 106
- estructura de datos, 177
  - genérica, 139, 169
- estructura de datos genérica, 139, 150
- excepción, 4, 8, 68
  - Target is Void, 78, 112
- explícito, 138
- expresión, 14, 15, 17, 21, 22, 96
  - lógica, 41
- expresiones, 154
- factor de carga, 210, 211
- factorial, 46
- feature, 7, 82, 159
  - {NONE}, 159
- FIFO, 160, 175
- first\_index\_of, 62
- forma contractual, 62
- forma corta, 62
- frabuloso, 44
- función, 87, 94
- función hash, 209–211
- función pura, 87, 129
- generalización, 54, 56, 58, 65, 75, 92
- genericidad, 169
- genericidad restringida, 175
- GOLFISTA, 172
- heapsort, 201, 202
- herencia, 127, 135
- histograma, 102, 105
- hola mundo, 6
- Holmes, Sherlock, 4
- HORA, 82
- identidad, 110, 134
- igual\_carta, 109
- igualdad, 110, 134, 208
- igualdad profunda, 110, 118, 210
- igualdad superficial, 110, 118
- implícito, 138
- implementación
  - ARBOL, 177, 184
  - Cola, 160
  - Cola con prioridad, 169, 196
  - PILA, 156
  - TABLA, 206
  - Tabla, 208
  - tabla hash, 208
- impresión, 7
- imprimir
  - array de CARTAs, 113
  - CARTA, 108
- imprimir\_baraja, 113, 120
- imprimir\_carta, 108
- incrementar, 65, 68
- index\_of, 62
- infija, 154
- infijo, 159, 180
- inorden, 180, 191
- instancia, 80, 94
- interface, 36
- interpretar, 2, 8
- invariante, 148, 150, 159
- invariante de objeto, 148
- is\_equal, 67, 134, 208
- item, 59
- iteración, 48, 58
- jerarquía de clases, 137
- lógico, 41, 43, 47
- langostinos, 97, 142
- lenguaje
  - alto nivel, 1
  - bajo nivel, 1
  - completo, 43
  - formal, 5
  - natural, 5, 109
  - programación, 1, 127
  - seguro, 4
- lenguaje de alto nivel, 1, 8
- lenguaje de bajo nivel, 1, 8
- lenguaje de programación, 1, 127
- lenguaje formal, 5, 8

- lenguaje natural, 5, 8, 109
- lenguaje seguro, 4
- Linux, 4
- lista, 139, 150
  - bien formada, 148
  - bucle, 143
  - como parámetro, 141
  - imprimir, 141
  - imprimir al revés, 143
  - infinita, 143
  - modificar, 145
  - recorrido, 142
  - recorrido recursivo, 142
- lista infinita, 143
- LISTA\_ENLAZADA, 147
- literalidad, 5
- logaritmo, 50
- método, 21, 25, 30, 55
  - auxiliar, 122, 126
  - ayudante, 146
  - cadena, 59
  - constructor, 81, 83
  - de acceso, 186
  - de creación, 81
  - definición, 23
  - envoltorio, 146
  - fructífero, 29
  - función, 37, 129
  - función pura, 87
  - invocar, 135
  - is\_equal, 134
  - lógico, 43
  - make, 23
  - modificador, 89, 131
  - objeto, 59, 127, 145
  - raíz, 81
  - rellenador, 90
  - to\_string, 132
  - varios parámetros, 28
  - Void, 37
- método auxiliar, 122, 126, 194
- método ayudante, 146
- método de cadena, 59
- método de creación, 81
- método de objeto, 59, 127, 138
- método envoltorio, 146
- método fructífero, 29
- método función, 37
- método normal, 138
- método raíz, 7, 81, 94
- método rellenador, 90
- métodos de acceso, 186
- módulo, 31, 209
- make, 7, 23
- manzanilla, 97, 142
- marco, 29
- marco de método, 29
- media, 99
- modificador, 89, 94, 131
- modificar listas, 145
- módulo, 36
- Montículo
  - análisis, 200
  - definición, 197
- montículo, 192
- mutable, 74
- número aleatorio, 99, 121
- número complejo, 128
- nivel, 178, 191
- no determinístico, 99
- nodo, 139, 150, 177
  - método de objeto, 145
- nodo de árbol, 177
- nodo hijo, 178, 191
- nodo hoja, 178, 191
- nodo padre, 178, 191
- nodo raíz, 178, 191
- notación O mayúscula, 201
- notación punto, 71, 131
- nueva línea, 9, 35
- objeto, 59, 68, 69, 87
  - actual, 128
  - array de, 112
  - como parámetro, 72
  - como tipo de retorno, 73
  - expandido, 79
  - imprimir, 86
  - mutable, 74
  - referencia, 79
- objeto actual, 128, 138
- objeto expandido, 79
- operador, 14, 17
  - cadena, 15
  - carácter, 65
  - comparación, 32
  - COMPLEJO, 129
  - división entera, 31
  - lógico, 42, 47
  - módulo, 31, 209
  - objeto, 87
  - punto, 127
  - relacional, 32, 41, 47, 110
- operador de cadena, 15
- operador lógico, 42
- operador punto, 127



- operador relacional, 32, 41, 110
- operando, 17
- orden, 106
- orden de crecimiento, 201, 202
- orden de las operaciones, 15
- ordenación, 124, 194, 201
- ordenación completa, 111
- ordenación parcial, 111
- ordenación por mezcla, 124, 202
- ordenación por selección, 122, 194, 202
- ordenar, 111, 122
- overloading, 83
  
- palabra clave, 14, 17
- palo, 106
- papel
  - variable, 144
- parámetro, 26, 30, 72
  - abstracto, 117
- parámetro abstracto, 117, 118
- parámetros
  - varios, 28
- PAR\_CLAVE\_VALOR, 206
- parse, 159
- PILA
  - implementación basada en array, 156
- pila, 35, 45, 151
- poesía, 5
- postcondición, 63, 159
- postcondiciones, 149
- postfija, 154
- postfijo, 159, 180
- postorden, 180, 191
- precondición, 63, 144, 150, 159
- precondiciones, 149
- predicado, 159
- prefijo, 191
- preorden, 180, 191
- print, 9, 86
- prioridad, 15
- programación
  - funcional, 127
  - imperativa, 127
  - orientada a objetos, 127
- programación funcional, 94, 127
- programación imperativa, 127
- programación orientada a objetos, 127
- propiedad de montículo, 197
- prosa, 5
- prototipado, 91
- proveedor, 151, 159, 182
- pseudoaleatorio, 105
- pseudocódigo, 121, 126, 195
- PUNTO, 70
  
- real, 18, 30
- recogida de basura, 79, 80
- recorrer, 61, 68, 142, 208
  - contar, 64
- recorrido, 114, 142, 179, 180, 205
  - array, 101
- RECTANGULO, 73, 136
- recursión
  - infinita, 117
- recursión infinita, 36, 117
- recursividad, 34, 36, 43, 115, 125, 180, 195, 200
  - infinita, 36
- recursivo, 35
- redimensionar, 210
- redimensionar una tabla hash, 210
- redondeo, 19
- redundancia, 5
- referencia, 70, 77, 79, 80, 108, 122, 123, 139
  - embebida, 139
- referencia embebida, 139
- referencia interna, 177
- require, 63
- resolución de problemas, 8
- restituir propiedad de montículo, 198
- Result, 73
- retorno, 37
- riesgo en eficiencia, 161, 175
  
- salto de fé, 45
- salto de fe, 126, 143
- semántica, 4, 8, 42
- sentencia, 2, 17
  - alternativa, 31
  - asignación, 11, 48
  - comentario, 7
  - create, 70, 85
  - declaración, 11, 70
  - impresión, 7
  - loop, 49
  - print, 9, 86
- sentencia loop, 49
- sintaxis, 3, 8
- sistema, 94
- sobrecarga, 196, 202
- STRING, 9, 66, 67, 69
  - array de, 108
  - count, 60
  - referencia a, 108
- sub-baraja, 117, 122
- suma desplazada, 209, 211
  
- tabla, 50, 203, 211
  - bidimensional, 53

- implementación basada en array, 206
- implementación de tabla hash, 208
- tabla hash, 211
  - implementación, 208
  - redimensionar, 210
- TAD, 151, 155, 159, 177, 192
  - Arbol, 184
  - Cola, 160
  - Cola con prioridad, 160, 168
  - PILA, 151
  - Tabla, 203
- TAD Cola, 160
- TAD Tabla, 203
- teorema
  - fundamental de la ambigüedad, 144
- test, 125
- tiempo constante, 175, 196, 201
- tiempo cuadrático, 194, 201
- tiempo de ejecución, 192
- tiempo lineal, 163, 175, 196, 201
- tiempo logarítmico, 201
- tipo, 17
  - array, 95
  - CHARACTER, 59, 65
  - definido por el usuario, 81
  - double, 18
  - objeto, 81, 134
  - STRING, 9, 69
- tipo abstracto de datos, 151
- tipo de objeto, 81
- tipo de retorno, 47
- tipo definido por el usuario, 81
- to\_lower, 66
- to\_string, 132
- to\_upper, 66
- token, 159
- transportable, 1
- trozo, 102
- Turing, Alan, 44, 67
- valor, 10, 17, 211
  - carácter, 60
- valor de retorno, 37, 47
- variable, 10, 17
  - atributo, 71
  - contador, 53, 56, 61, 97
  - local, 56, 58
  - papeles, 144
  - Result, 37, 73, 114
- variable contador, 56, 61, 97
- variable local, 56, 58
- varias asignaciones, 48
- verificación, 117
- VISITABLE, 182
- Void, 37, 78, 95, 112