

Pensando la computación como un científico (con Java)

Allen B. Downey

Traducción al español (Argentina):
Diego Delle Donne, Germán Kruszewski,
Francisco Laborda, Maximiliano Llosa y Javier Pimás



Universidad
Nacional de
General
Sarmiento

Índice general

Prefacio	VII
1 El camino del programa	1
1.1 ¿Qué es un lenguaje de programación?	1
1.2 ¿Qué es un programa?	4
1.3 ¿Qué es la depuración?	5
1.3.1 Errores de sintaxis	5
1.3.2 Errores en tiempo de ejecución	5
1.3.3 Errores semánticos	6
1.3.4 Depuración experimental	6
1.4 Lenguajes formales y lenguajes naturales	7
1.5 El primer programa	9
1.6 Glosario	10
1.7 Ejercicios	12
2 Variables y tipos	15
2.1 Imprimiendo más	15
2.2 Variables	16
2.3 Asignación	17
2.4 Imprimiendo variables	18
2.5 Palabras reservadas	20
2.6 Operadores	20
2.7 Orden de las operaciones	21
2.8 Operadores de cadenas	22
2.9 Composición	22
2.10 Glosario	23
2.11 Ejercicios	24
3 Métodos	27
3.1 Punto flotante	27
3.2 Convirtiendo entre double e int	28
3.3 Métodos de la clase Math	29

3.4	Composición	30
3.5	Agregando nuevos métodos	31
3.6	Clases y métodos.	33
3.7	Programas con múltiples métodos.	34
3.8	Parámetros y argumentos	35
3.9	Diagramas de la pila de ejecución	37
3.10	Métodos con múltiples parámetros	37
3.11	Métodos con resultados.	38
3.12	Glosario	38
3.13	Ejercicios	39
4	Condicionales y recursión	43
4.1	El operador módulo	43
4.2	Ejecución condicional	43
4.3	Ejecución alternativa	44
4.4	Condicionales encadenados	45
4.5	Condicionales anidados.	46
4.6	La sentencia return	46
4.7	Conversión de tipos.	47
4.8	Recursión.	47
4.9	Diagramas de la pila de ejecución para métodos recursivos	49
4.10	Convención y Ley Divina	50
4.11	Glosario	51
4.12	Ejercicios	52
5	Métodos con resultados	55
5.1	Valores de retorno	55
5.2	Desarrollo de un programa	57
5.3	Composición	59
5.4	Sobrecarga.	60
5.5	Expresiones booleanas	61
5.6	Operadores lógicos	62
5.7	Métodos booleanos	63
5.8	Más recursión	64
5.9	Salto de fe	66
5.10	Un ejemplo más	67
5.11	Glosario	68
5.12	Ejercicios	69

6	Iteración	75
6.1	Asignación múltiple	75
6.2	Iteración	76
6.3	La sentencia while	76
6.4	Tablas	78
6.5	Tablas de dos dimensiones	80
6.6	Encapsulamiento y generalización	81
6.7	Métodos	82
6.8	Más encapsulamiento	83
6.9	Variables locales	83
6.10	Más generalización	84
6.11	Glosario	86
6.12	Ejercicios	86
7	Cadenas y cosas	91
7.1	Invocando métodos en objetos	91
7.2	Largo	92
7.3	Recorrido	93
7.4	Errores en tiempo de ejecución	93
7.5	Leyendo la documentación	94
7.6	El método indexOf	95
7.7	Iterando y contando	96
7.8	Operadores de incremento y decremento	96
7.9	Los Strings son inmutables	97
7.10	Los Strings no son comparables	98
7.11	Glosario	99
7.12	Ejercicios	99
8	Objetos interesantes	105
8.1	¿Qué es interesante?	105
8.2	Paquetes	105
8.3	Objetos Point	106
8.4	Variables de instancia	107
8.5	Objetos como parámetros	107
8.6	Rectángulos	108
8.7	Objetos como tipo de retorno	109
8.8	Los objetos son mutables	109
8.9	Aliasing	110
8.10	null	111
8.11	Recolector de basura	112
8.12	Objetos y tipos primitivos	113

8.13	Glosario	113
8.14	Ejercicios	114
9	Creá tus propios objetos	119
9.1	Definiciones de clases y tipos de objetos	119
9.2	Tiempo	120
9.3	Constructores	121
9.4	Más constructores	122
9.5	Creando un objeto nuevo	122
9.6	Imprimiendo un objeto	124
9.7	Operaciones sobre objetos	124
9.8	Funciones puras	125
9.9	Modificadores	127
9.10	Métodos de llenado	128
9.11	¿Cuál es mejor?	128
9.12	Desarrollo incremental vs. planificación	129
9.13	Generalización	130
9.14	Algoritmos	131
9.15	Glosario	131
9.16	Ejercicios	132
10	Arreglos	137
10.1	Accediendo a los elementos	138
10.2	Copiando arreglos	139
10.3	Ciclos for	139
10.4	Arreglos y objetos	140
10.5	Longitud de un arreglo	141
10.6	Números aleatorios	141
10.7	Arreglos de números aleatorios	142
10.8	Contando	144
10.9	El histograma	145
10.10	Solución de una sola pasada	146
10.11	Glosario	146
10.12	Ejercicios	147
11	Arreglos de Objetos	153
11.1	Composición	153
11.2	Objetos Carta	153
11.3	El método imprimirCarta	155
11.4	El método mismaCarta	157
11.5	El método compararCarta	158

11.6	Arreglos de cartas	159
11.7	El método imprimirMazo	161
11.8	Búsqueda	161
11.9	Mazos y submazos.	165
11.10	Glosario	166
11.11	Ejercicios	166
12	Objetos como Arreglos	169
12.1	La clase Mazo	169
12.2	Mezclando	171
12.3	Ordenamiento	172
12.4	Submazos	172
12.5	Mezclando y repartiendo	174
12.6	Mergesort	174
12.7	Glosario	177
12.8	Ejercicios	177
13	Programación Orientada a Objetos	181
13.1	Lenguajes de programación y estilos.	181
13.2	Métodos de clase y de objeto.	182
13.3	El objeto actual	182
13.4	Números complejos	182
13.5	Una función de números Complejos	183
13.6	Otra función sobre números Complejos	184
13.7	Un modificador	185
13.8	El método toString	186
13.9	El método equals	187
13.10	Llamando a un método de objeto desde otro	188
13.11	Rarezas y errores.	188
13.12	Herencia	189
13.13	Rectángulos dibujables	189
13.14	La jerarquía de clases	190
13.15	Diseño orientado a objetos	191
13.16	Glosario	191
13.17	Ejercicios	191
14	Listas enlazadas	193
14.1	Referencias en objetos.	193
14.2	La clase Nodo	193
14.3	Listas como colecciones.	195
14.4	Listas y recursión	196

14.5	Listas infinitas	197
14.6	El teorema fundamental de la ambigüedad	198
14.7	Métodos de instancia para nodos	199
14.8	Modificando listas	199
14.9	Adaptadores y auxiliares	200
14.10	La clase <code>ListaInt</code>	201
14.11	Invariantes	202
14.12	Glosario	203
14.13	Ejercicios	203
15	Pilas	205
15.1	Tipos de datos abstractos	205
15.2	El TAD Pila	206
15.3	El objeto <code>Stack</code> de Java	206
15.4	Clases adaptadoras	208
15.5	Creando adaptadores	209
15.6	Creando más adaptadores	209
15.7	Sacando los valores afuera.	209
15.8	Métodos útiles en las clases adaptadoras	210
15.9	Notación polaca	211
15.10	Parseo	211
15.11	Implementando TADs	213
15.12	Implementación del TAD Pila usando arreglos	213
15.13	Redimensionando el arreglo	214
15.14	Glosario	216
15.15	Ejercicios	217
16	Colas y colas de prioridad	221
16.1	El TAD Cola	222
16.2	Veneer	223
16.3	Cola enlazada	224
16.4	Buffer circular	226
16.5	Cola de prioridad	230
16.6	Metacalse	231
16.7	Implementación de Cola de Prioridad sobre Arreglo	231
16.8	Un cliente de la Cola de Prioridad	233
16.9	La clase <code>Golfista</code>	234
16.10	Glosario	236
16.11	Ejercicios	237

17 Árboles	239
17.1 Un nodo del árbol	239
17.2 Construcción de árboles	240
17.3 Recorrido de árboles	240
17.4 Árboles de expresiones	241
17.5 Recorrido	242
17.6 Encapsulamiento	243
17.7 Definición de una metaclase	244
17.8 Implementando una metaclase	245
17.9 La clase Vector	246
17.10 La clase Iterator	248
17.11 Glosario	249
17.12 Ejercicios	250
18 Heap	253
18.1 Implementación de árbol sobre arreglo	253
18.2 Análisis de eficiencia	257
18.3 Análisis de mergesort	259
18.4 Overhead	261
18.5 Implementaciones de la Cola de Prioridad	261
18.6 Definición de un Heap	263
18.7 quitar en Heap	264
18.8 agregar en Heap	266
18.9 Eficiencia de los Heaps	266
18.10 Heapsort	267
18.11 Glosario	268
18.12 Ejercicios	269
19 Maps	271
19.1 Arreglos, Vectores y Maps	271
19.2 El TAD Map	272
19.3 El HashMap preincorporado	272
19.4 Una implementación usando Vector	274
19.5 La metaclase List	277
19.6 Implementación de HashMap	277
19.7 Funciones de Hash	278
19.8 Redimensionamiento de un HashMap	279
19.9 Rendimiento del redimensionado	280
19.10 Glosario	280
19.11 Ejercicios	281

20	Código Huffman	285
20.1	Códigos de longitud variable	285
20.2	La tabla de frecuencias	286
20.3	El árbol de Huffman.	287
20.4	El método super	290
20.5	Decodificando	291
20.6	Codificando	292
20.7	Glosario	293
	Apéndices	295
A	Planificación del desarrollo de un programa	295
B	Depuración	303
B.1	Errores en tiempo de compilación	303
B.2	Errores en tiempo de ejecución	307
B.3	Errores semánticos	310
C	Entrada y salida en Java	317
C.1	Objetos del sistema	317
C.2	Entrada por teclado	318
C.3	Entrada por archivos	318
D	Gráficos	321
D.1	Objetos Pizarra y Graphics	321
D.2	Llamando métodos en un objeto Graphics	322
D.3	Coordenadas	323
D.4	Un Ratón Mickey medio pelo	324
D.5	Otros comandos para dibujar.	325
D.6	La clase Pizarra	326
E	Licencia de Documentación Libre de GNU	333
E.1	Aplicabilidad y definiciones.	334
E.2	Copia literal.	335
E.3	Copiado en cantidades	335
E.4	Modificaciones.	336
E.5	Combinando documentos	338
E.6	Colecciones de documentos.	338
E.7	Agregación con trabajos independientes	338
E.8	Traducción.	339
E.9	Terminación.	339
E.10	Futuras revisiones de esta licencia	339
E.11	Addendum.	340

Capítulo 1

El camino del programa

El objetivo de este libro es enseñarte a pensar como lo hacen los científicos informáticos. Esta manera de pensar combina algunas de las mejores características de la matemática, la ingeniería y las ciencias naturales. Como los matemáticos, los científicos informáticos usan lenguajes formales para denotar ideas (específicamente, cómputos). Como los ingenieros, diseñan cosas, construyendo sistemas mediante el ensamble de componentes y evaluando las ventajas y desventajas de cada una de las alternativas de construcción. Como los científicos, observan el comportamiento de sistemas complejos, forman hipótesis, y prueban sus predicciones.

La habilidad más importante del científico informático es la **resolución de problemas**. La resolución de problemas incluye poder formular problemas, pensar en soluciones de manera creativa, y expresar soluciones claras y precisas. Como veremos, el proceso de aprender a programar es la oportunidad perfecta para desarrollar la habilidad de resolver problemas. Por esa razón este capítulo se llama “El camino del programa”.

En cierta medida, aprenderemos a programar, lo cual es una habilidad muy útil por sí misma. Por otra parte, utilizaremos la programación para obtener algún resultado. Ese resultado se verá más claramente durante el proceso.

1.1 ¿Qué es un lenguaje de programación?

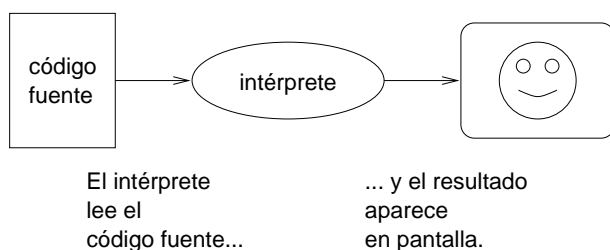
El lenguaje que aprenderás es Java, que es relativamente nuevo (Sun liberó la primera versión en mayo de 1995). Java es un ejemplo de un **lenguaje de alto nivel**; otros lenguajes de alto nivel de los que habrás oído hablar son Pascal, C, C++, Python, PHP, entre otros.

Como se puede deducir, además de “lenguajes de alto nivel” también existen **lenguajes de bajo nivel**, que también se denominan “lenguajes de máquina” o “lenguajes ensambladores.” A propósito, las computadoras sólo ejecutan programas escritos en lenguajes de bajo nivel. Así, los programas escritos en lenguajes de alto nivel tienen que ser traducidos antes de ser ejecutados. Esta traducción lleva tiempo, lo que es una pequeña desventaja de los lenguajes de alto nivel.

Aun así, las ventajas son enormes. En primer lugar, la programación en lenguajes de alto nivel es **mucho** más fácil; escribir programas en un lenguaje de alto nivel toma menos tiempo, los programas son más cortos y más fáciles de leer, y es más probable que estos programas sean correctos. En segundo lugar, los lenguajes de alto nivel son **portables**, lo que significa que los programas escritos con estos lenguajes pueden ser ejecutados en diferentes tipos de computadoras sin modificación alguna o con pocas modificaciones. Los programas escritos en lenguajes de bajo nivel sólo pueden ser ejecutados en un tipo de computadora y deben ser reescritos para ser ejecutados en otra.

Debido a estas ventajas, casi todo programa se escribe en un lenguaje de alto nivel. Los lenguajes de bajo nivel son sólo usados para unas pocas aplicaciones especiales.

Hay dos tipos de programas que traducen lenguajes de alto nivel a lenguajes de bajo nivel: **intérpretes** y **compiladores**. Un intérprete lee un programa de alto nivel y lo ejecuta, lo que significa que lleva a cabo lo que indica el programa. Es decir, traduce el programa poco a poco, leyendo y ejecutando cada comando.

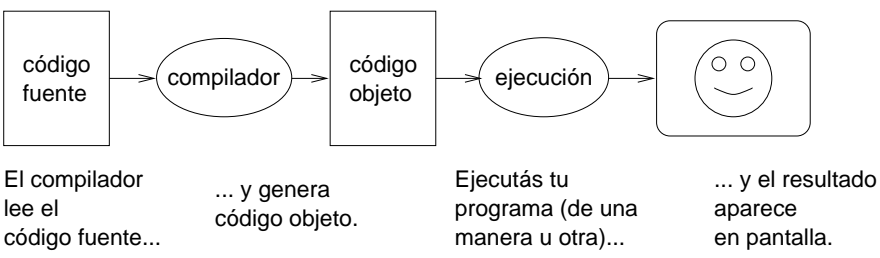


En cambio, un compilador lee el programa y lo traduce completo antes de su ejecución. En este caso, al programa de alto nivel se le llama **código fuente**, y el programa traducido es llamado **código objeto** o **programa ejecutable**. Una vez que un programa ha sido compilado, puede ser ejecutado repetidamente sin necesidad de más traducción.

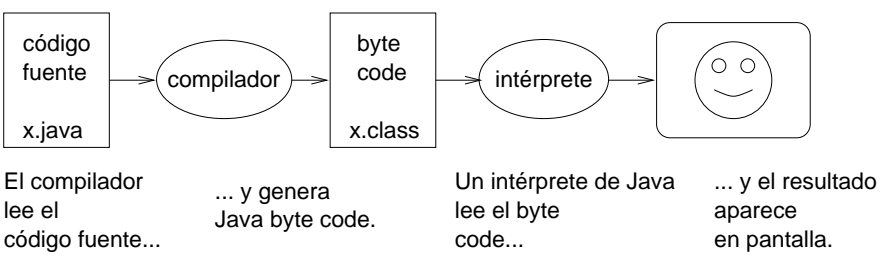
Por ejemplo, supongamos que escribís un programa en C utilizando un editor de texto (el bloc de notas, por ejemplo, sería suficiente). El pro-

grama terminado lo podrías guardar en un archivo llamado `programa.c`, donde “programa” es un nombre arbitrario que inventás y la extensión `.c` es una convención que indica que el archivo contiene código fuente de C.

A continuación, corrés el compilador sobre ese código fuente. El compilador leerá el código, lo traducirá (al lenguaje de bajo nivel apropiado para la máquina), y creará un nuevo archivo llamado `programa.o` que contiene el código objeto, o `programa.exe` que contiene el programa ejecutable.



El lenguaje Java es excepcional porque es tanto compilado como interpretado. En lugar de traducir los programas Java en lenguaje de máquina, el compilador de Java genera *Java byte code*. *Byte code* es un lenguaje veloz y fácil de interpretar por la computadora —como los lenguajes de máquina—, pero es también portable —como los lenguajes de alto nivel. Por lo tanto, es posible compilar un programa Java en una máquina, transferir el *byte code* generado a otra máquina, y luego ejecutar el programa en esa otra máquina, a pesar de que ésta pueda tener otro sistema operativo. Esta es una de las ventajas de Java frente a otros lenguajes de alto nivel.



A pesar de que este proceso pueda parecer complicado, en la mayoría de los entornos de programación (a veces llamados entornos de desarrollo), estos pasos se realizan automáticamente. Generalmente alcanza solamente con escribir el programa y presionar algún botón con el cual el

programa se compila y se ejecuta sin que nosotros tengamos que preocuparnos por cómo se hace este proceso. Sin embargo, es útil que conozcas cuáles son los pasos que se dan sin intervención, de modo que si algo sale mal, podés intentar deducir dónde estuvo el problema.

1.2 ¿Qué es un programa?

Un programa es una secuencia de instrucciones que especifica cómo efectuar un cálculo o cómputo. El cómputo puede ser algo matemático, como resolver un sistema de ecuaciones o encontrar la raíz de un polinomio; pero también puede ser simbólico, como buscar y reemplazar texto en un documento o (aunque parezca raro) compilar un programa.

Las **sentencias** o instrucciones tienen una apariencia diferente en lenguajes de programación distintos, pero existen algunas que son básicas, que se presentan en casi todo lenguaje, y que pueden agruparse en los siguientes conceptos:

entrada: Reciben datos del teclado, de un archivo o de algún otro dispositivo.

salida: Muestran datos en el monitor o los envían a un archivo u otro dispositivo.

matemáticas: Ejecutan operaciones básicas de matemáticas como la adición y la multiplicación.

operación condicional: Prueban la veracidad de alguna condición y ejecutan la secuencia de instrucciones apropiada.

repetición: Ejecutan alguna acción repetidas veces, usualmente con alguna variación.

Aunque sea difícil de creer, estos conceptos agrupan más o menos todas las instrucciones que hay en los lenguajes de programación. Todos los programas en existencia, sin importar su complejidad, son formulados exclusivamente con tales instrucciones. Así, la programación se puede describir como el proceso de dividir una tarea amplia y compleja en tareas cada vez más pequeñas hasta que éstas sean lo suficientemente sencillas como para ser ejecutadas con estas instrucciones básicas.

Quizás esta descripción sea un poco imprecisa, pero volveremos a este tema más adelante cuando hablemos de algoritmos.

1.3 ¿Qué es la depuración?

La programación es un proceso complejo y, debido a que es realizada por seres humanos, a veces conlleva la aparición de errores. Por caprichos del destino, estos errores de programación se denominan **bugs**¹ y el proceso de buscarlos y corregirlos es llamado **depuración** o *debugging*.

Hay tres tipos de errores que pueden ocurrir en un programa: errores de sintaxis, errores en tiempo de ejecución y errores semánticos. Es muy útil distinguirlos para encontrarlos más rápido.

1.3.1 Errores de sintaxis

El compilador sólo puede traducir un programa cuando es sintácticamente correcto. De lo contrario, la compilación falla y desde luego, no se podrá ejecutar el programa. **Sintaxis** es la estructura del programa y las reglas que rigen dicha estructura. Por ejemplo, en español, una oración debe comenzar con mayúscula y terminar con punto. esta oración contiene un error de sintaxis. Esta también

Para la mayoría de los lectores, unos pocos errores de sintaxis no son un problema significativo, razón por la cual podemos leer las obras de Roberto Arlt sin proferir algún tipo de mensaje de error.

Los compiladores no son tan indulgentes. Si hay el más mínimo error de sintaxis en algún lugar del programa, el compilador mostrará un mensaje de error y abortará la tarea, por lo cual, no se generará el programa ejecutable.

Al principio, los errores de sintaxis son bastante comunes, pero a medida que vayamos tomando experiencia, tendremos menos y los encontraremos cada vez más rápidamente.

1.3.2 Errores en tiempo de ejecución

El segundo tipo de error es el error en tiempo de ejecución, llamado así porque sólo aparece cuando se ejecuta un programa. En Java, los errores en tiempo de ejecución aparecen cuando el intérprete está ejecutando el *byte code* y algo sale mal. Por ejemplo si al efectuar una división, dividimos por 0, obtendremos uno de estos errores.

En los primeros capítulos, sin embargo, este tipo de errores no serán muy comunes. Más tarde aparecerán con mayor frecuencia, especialmente cuando comencemos a hablar de objetos y referencias (Capítulo 8).

En Java, los errores de ejecución se llaman **excepciones**, y en la mayoría de los entornos aparecen como ventanas que contienen información de qué ocurrió y qué estaba haciendo el programa cuando sucedió. Esta información es útil para la depuración.

1. N.d.T.: ‘bichos’ en inglés.

1.3.3 Errores semánticos

El tercer tipo de error es el **error semántico**. Si hay un error semántico en su programa, el programa será ejecutado sin ningún mensaje de error, pero el resultado no será el deseado. El programa ejecutará exactamente lo que le dijiste que ejecutara.

A veces ocurre que el programa escrito no es el programa que se tenía en mente. El sentido o significado del programa (su valor semántico) no es correcto. Identificar errores semánticos es difícil porque requiere trabajar al revés: comenzar por los resultados de salida y tratar de descifrar lo que el programa está realizando.

1.3.4 Depuración experimental

Una de las técnicas más importantes que aprenderás es la depuración. Aunque a veces es frustrante, la depuración es una de las partes de la programación intelectualmente más exigentes, desafiantes e interesantes.

La depuración es una actividad parecida a la labor realizada por detectives: se tienen que estudiar las pistas para inferir los procesos y eventos que han generado los resultados que se han encontrado.

La depuración es como una ciencia experimental. Una vez que tenés una idea de qué está saliendo mal, modificás el programa e intentás nuevamente. Si la hipótesis fue la correcta, entonces podés predecir los resultados de la modificación y estarás más cerca a un programa correcto. Si la hipótesis fue errónea tendrás que idearte otra. Como dijo Sherlock Holmes, “Cuando se ha descartado lo imposible, lo que queda, no importa cuán inverosímil, debe ser la verdad”².

Para algunas personas, la programación y la depuración son lo mismo: la programación es el proceso de depurar un programa gradualmente hasta que el programa realice lo deseado. Esto quiere decir que el programa debe ser, desde el principio, un programa que funcione y que realice *algo*; a este programa se le hacen pequeñas modificaciones y se lo depura manteniéndolo siempre funcionando.

Por ejemplo, aunque el sistema operativo Linux contenga miles de líneas de instrucciones, Linus Torvalds lo comenzó como un programa para explorar el microprocesador Intel 80386. Según Larry Greenfield, “Uno de los primeros proyectos de Linus fue un programa que intercambiaría la impresión de AAAA y BBBB. Este programa se convirtió en Linux”³.

En capítulos posteriores se tratará más el tema de la depuración y de otras técnicas de programación.

2. A. Conan Doyle, *The Sign of Four*.

3. *The Linux Users' Guide* Versión Beta 1.

1.4 Lenguajes formales y lenguajes naturales

Los **lenguajes naturales** son los lenguajes hablados por seres humanos, como el español, el inglés y el francés. Estos no han sido diseñados artificialmente (aunque se trate de imponer cierto orden en ellos), sino que se han desarrollado naturalmente.

Los **lenguajes formales** son diseñados por seres humanos para aplicaciones específicas. La notación matemática, por ejemplo, es un lenguaje formal, ya que se presta a la representación de las relaciones entre números y símbolos. Los químicos utilizan un lenguaje formal para representar la estructura química de las moléculas. Y lo más importante:

Los lenguajes de programación son lenguajes formales que han sido desarrollados para expresar cómputos.

Como mencioné antes, los lenguajes formales casi siempre tienen reglas sintácticas estrictas. Por ejemplo, $3 + 3 = 6$ es una expresión matemática sintácticamente correcta, pero $3 = +6\$$ no lo es. De la misma manera, H_2O es una nomenclatura química sintácticamente correcta, pero $_2Zz$ no lo es. Existen dos clases de reglas sintácticas: componentes léxicos⁴ y estructura. Los componentes léxicos son los elementos básicos de un lenguaje, como lo son las palabras, los números y los elementos químicos. Por ejemplo, en $3 = +6$, $\$$ no es (hasta donde sabemos) un símbolo matemático aceptado. De manera similar, $_2Zz$ no es válido porque no hay ningún elemento químico con la abreviación Zz .

La segunda clase de regla sintáctica está relacionada con la estructura de una sentencia; es decir, con el orden de los componentes léxicos. La estructura de la sentencia $3=+6\$$ no es válida porque no se puede escribir el símbolo de igualdad seguido del signo de la adición. Similarmente, las fórmulas químicas tienen que mostrar el número de subíndice después del elemento, no antes.

Al leer una oración en castellano o una sentencia en un lenguaje formal, se debe discernir la estructura de la oración (aunque en el caso de un lenguaje natural, lo hacemos inconscientemente). Este proceso se conoce como **análisis sintáctico**⁵.

Por ejemplo cuando se escucha una oración simple como “el pez por la boca muere”, se puede distinguir que “el pez” es el sujeto, “muere” es el verbo y “por la boca” un modificador del verbo. Cuando se ha analizado

4. N.d.T: En inglés se denominan *tokens* y pueden aparecer frecuentemente con este nombre.

5. N.d.T.: En inglés, *parsing*. Asimismo, los analizadores sintácticos se llaman *parsers*.

la oración sintácticamente, se puede deducir el significado, es decir, la semántica, de la oración. Si sabés lo que es un pez, una boca y el significado de morir, vas a comprender el significado de la oración.

Aunque existen muchas cosas en común entre los lenguajes naturales y los lenguajes formales — por ejemplo los componentes léxicos, la estructura, la sintaxis y la semántica — también existen muchas diferencias:

Ambigüedad: Los lenguajes naturales tienen muchísimas ambigüedades que se superan usando claves contextuales e información adicional. Los lenguajes formales son diseñados para estar completamente libres de ambigüedades, tanto como sea posible, lo que quiere decir que cualquier sentencia tiene sólo un significado sin importar el contexto en el que se encuentra.

Redundancia: Para reducir la ambigüedad y los malentendidos, los lenguajes naturales utilizan bastante redundancia. Como resultado tienen una abundancia de posibilidades para expresarse. Los lenguajes formales son menos redundantes y más concisos.

Literalidad: Los lenguajes naturales tienen muchas metáforas y frases comunes. El significado de un dicho, por ejemplo “el pez por la boca muere”, es diferente al significado de sus sustantivos y verbos. En este ejemplo, la oración no tiene nada que ver con un pez y significa que habló de más. En los lenguajes formales sólo existe el significado literal.

Los que aprenden a hablar un lenguaje natural —o sea, casi todo el mundo— muchas veces tienen dificultad en adaptarse a los lenguajes formales. A veces la diferencia entre los lenguajes formales y los naturales es comparable a la diferencia entre la prosa y la poesía:

Poesía: Se utiliza la palabra por su cualidad auditiva tanto como por su significado. El poema, en su totalidad, produce un efecto o reacción emocional. La ambigüedad no es sólo común sino utilizada a propósito.

Prosa: El significado literal de la palabra es más importante y la estructura contribuye aún más al significado. La prosa se presta más al análisis que la poesía pero todavía contiene ambigüedad.

Programas: El significado de un programa es inequívoco y literal, y es entendido en su totalidad a través del análisis de las unidades léxicas y la estructura.

He aquí unas sugerencias para la lectura de un programa (y de otros lenguajes formales). Primero, recordá que los lenguajes formales son mucho más densos que los lenguajes naturales, y en consecuencia toma más tiempo dominarlos. Además, la estructura es muy importante, por lo cual usualmente no es una buena idea leerlo de arriba a abajo y de izquierda a derecha. En lugar de esto, aprendé a separar las diferentes partes en tu mente, identificar los componentes léxicos e interpretar la estructura. Finalmente, poné atención a los detalles. Los errores de puntuación y la ortografía, que en los lenguajes naturales podemos aún comprender un texto a pesar de ellos, en los lenguajes formales trazan una gruesa línea entre lo correcto y lo incorrecto.

1.5 El primer programa

Tradicionalmente el primer programa que la gente escribe en un lenguaje nuevo se llama “Hola, mundo” porque lo único que hace es mostrar en pantalla las palabras “Hola, mundo”. En Java, este programa es así:

```
class Hola{

    // main: mostrar una salida sencilla

    public static void main (String[] args) {
        System.out.println ("Hola, mundo.");
    }
}
```

Algunos juzgan la calidad de un lenguaje de programación por la simplicidad del programa “Hola, mundo”. Mirado con este estándar, Java no se desempeña muy bien. Incluso el programa más simple contiene una cantidad de características que son difíciles de explicar a los programadores principiantes. Ignoraremos muchas por ahora, pero explicaré alguna de ellas. Todos los programas están conformados por *definiciones de clases*, que tienen la forma:

```
class NOMBRE_CLASE {

    public static void main (String[] args) {
        SENTENCIAS
    }
}
```

Aquí NOMBRE_CLASE indica un nombre arbitrario que inventás. El nombre de la clase en el ejemplo es Hola.

En la segunda línea, ignoraré las palabras `public static void` por ahora, pero observá la palabra `main`. `main` es un nombre especial que indica el punto en el cual se inicia la ejecución del programa. Cuando el programa se ejecuta, comienza por ejecutar la primera sentencia en `main` y continúa, en orden, hasta que llega a la última sentencia, para luego finalizar.

No existe ningún límite a la cantidad de sentencias que pueden aparecer en el `main`, pero el ejemplo contiene sólo una. Es una **sentencia `print`**, que se utiliza para “imprimir”⁶ un mensaje en pantalla. Es un poco confuso que “imprimir” a veces significa “mostrar algo en pantalla”, y a veces “mandar algo a la impresora”. En este libro no diré mucho acerca de enviar cosas a la impresora, con lo cual, todas las veces que hablemos de impresiones serán en pantalla.

La sentencia que imprime valores por pantalla es `System.out.println`, y lo que aparece entre paréntesis es el valor que será impreso. Al final de la sentencia hay un punto y coma (;), el cual es necesario al final de toda sentencia.

Hay algunas otras cosas que deberías observar sobre la sintaxis de este programa. Primero, que Java utiliza llaves({ y }) para agrupar cosas. Las llaves externas (líneas 1 y 8) contienen la definición de la clase, y las interiores contienen la definición del `main`.

Además, notá que la línea 3 comienza con `//`. Esto indica que dicha línea contiene un **comentario**, que es, de algún modo, texto en castellano que se pone en distintas partes del programa, generalmente para explicar qué es lo que hace. Cuando el compilador ve una sentencia comenzada por `//`, ignora todo lo que aparezca desde ahí hasta el fin de la línea.

1.6 Glosario

resolución de problemas: Proceso de formular un problema, encontrar una solución, y expresarla.

lenguaje de alto nivel: Lenguaje de programación, al estilo Java, diseñado para ser fácil de leer y escribir por humanos.

lenguaje de bajo nivel: Lenguaje de programación diseñado para ser fácil de ejecutar por una computadora. También se conoce como “lenguaje de máquina” o “lenguaje ensamblador”.

lenguaje formal: Cualquier lenguaje diseñado que tiene un propósito específico, como la representación de ideas matemáticas o programas de computadoras; todos los lenguajes de programación son lenguajes formales.

6. N.d.T.: “imprimir” se dice “print” en inglés.

lenguaje natural: Cualquier lenguaje hablado que evolucionó de forma natural

portabilidad: Cualidad de un programa que puede ser ejecutado en más de un tipo de computadora.

interpretar: Ejecutar un programa escrito en un lenguaje de alto nivel traduciéndolo línea por línea.

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

código fuente: Programa escrito en un lenguaje de alto nivel antes de ser compilado.

código objeto: Salida del compilador una vez que el programa ha sido traducido.

ejecutable: Código objeto que está listo para ser ejecutado.

byte code: Tipo especial de código objeto utilizado por programas Java. *Byte code* es una especie de lenguaje de bajo nivel, pero portable, como los lenguajes de alto nivel.

sentencia: Parte de un programa que especifica un acción que se efectuará cuando el programa se ejecute. Una sentencia `print`, por ejemplo, hace que se muestre un valor por pantalla.

comentario: Parte del programa que contiene información para ser leída por una persona, pero que no tiene efecto en la ejecución del programa.

algoritmo: Proceso general para resolver un tipo particular de problemas.

bug: Error en un programa.

sintaxis: Estructura de un programa.

semántica: Significado de un programa.

análisis sintáctico: Examinar y analizar la estructura sintáctica de un programa.

error sintáctico: Error en un programa que lo hace imposible de analizar sintácticamente, y por lo tanto, imposible de compilar.

error en tiempo de ejecución: Error que se produce durante la ejecución de un programa.

excepción: Otro nombre para un error en tiempo de ejecución.

error semántico o lógico: Error en un programa por el cual hace algo diverso de lo que el programador pretendía.

depuración: Proceso de hallazgo y eliminación de los tres tipos de errores de programación.

1.7 Ejercicios

Ejercicio 1.1

Los científicos de la computación tienen el hábito molesto de usar palabras del castellano para referirse a cosas que no tienen nada que ver con su significado más común. Por ejemplo, en castellano, una sentencia es la disposición de un juez, mientras que en programación, es básicamente una instrucción.

El glosario al final de cada capítulo pretende resaltar palabras y frases que tienen un significado especial en las ciencias de la computación. Cuando veas palabras conocidas en el contexto de la programación, ¡no asumas que sabes su significado!

- a. ¿Qué es una sentencia? ¿Qué es un comentario?
- b. ¿Qué quiere decir que un programa sea portable?
- c. ¿Qué es compilar un programa? ¿A qué nos referimos cuando decimos que un lenguaje es interpretado?

Ejercicio 1.2

Antes que nada, descubrí cómo compilar y ejecutar un programa Java en tu entorno de desarrollo. Algunos entornos proveen programas de ejemplo similares al visto en la sección 1.5.

- a. Escribí el programa “Hola mundo”, luego compilalo y ejecutalo.
- b. Agregá una segunda sentencia print que imprima otro mensaje después del “¡Hola mundo!”. Algo ingenioso como “Cómo estás?”. Compilá y ejecutá el programa de nuevo.
- c. Agregá una línea de comentario (en cualquier parte) y recompilá y ejecutá el programa nuevamente. La nueva línea no debería afectar la ejecución del mismo.

Este ejercicio puede parecer trivial, pero es el punto de inicio para muchos de los programas con los que trabajaremos. Para poder depurar con confianza, debés tener seguridad sobre el entorno con el que programás. En algunos entornos, es

fácil perder la noción de qué programa es el que se está ejecutando y podés encontrarte intentando depurar un programa mientras que accidentalmente, estás ejecutando otro. Agregar (y cambiar) impresiones en pantalla es una manera simple de establecer una conexión entre el programa que estás mirando y la salida que muestra cuando se ejecuta.

Ejercicio 1.3

Una buena idea es cometer tantos errores como se te ocurran, de modo que puedas ver los mensajes que produce el compilador. A veces el compilador te dirá exactamente qué es lo que está mal, y todo lo que tenés que hacer es arreglarlo. A veces, sin embargo, el compilador producirá errores engañosos. Poco a poco irás desarrollando una intuición de cuándo podés confiar en el compilador y cuándo tenés que descubrir el problema por cuenta propia.

- a. Quitar una de las llaves que abren (`{`).
- b. Quitar una de las llaves que cierran (`}`).
- c. En lugar de `main`, escribí `main`.
- d. Quitar la palabra `static`.
- e. Quitar la palabra `public`.
- f. Quitar la palabra `System`.
- g. Reemplazar `println` por `println`.
- h. Reemplazar `println` por `print`. Este es un poco engañoso porque no es un error de sintaxis, sino uno de semántica. La sentencia `System.out.print` es válida, pero puede o puede no ser lo que esperás.
- i. Borrá uno de los paréntesis. Agregá uno extra.

Capítulo 2

Variables y tipos

2.1 Imprimiendo más

Como he mencionado en el último capítulo, en el main podés poner tantas sentencias como quieras. Por ejemplo, para imprimir más de una línea:

```
class Hola {  
  
    // main: genera una salida simple  
  
    public static void main (String[] args) {  
        System.out.println ("Hola, mundo."); // imprime una línea  
        System.out.println ("¿Cómo estás?"); // imprime otra  
    }  
}
```

Además, como se puede ver, es correcto dejar comentarios al final de una línea, como así también dentro de la línea en sí.

Las frases que aparecen entre comillas se llaman **cadenas**¹, porque están hechas de secuencias (o cadenas) de letras. En realidad, las cadenas pueden contener cualquier combinación de letras, números, signos de puntuación, y otros caracteres especiales.

`println` es la abreviatura de “print line” (“imprimir línea”) porque luego de cada línea, agrega un carácter especial, llamado **newline** (nueva línea), que mueve el cursor a la siguiente línea de la pantalla. La próxima vez que se llame a `println`, el nuevo texto aparecerá en la siguiente línea.

1. N.d.T.: En inglés se llaman “strings”, nombre que vas a encontrar muy seguido.

A menudo es útil mostrar la salida de múltiples sentencias de impresión todas en una línea. Puedes hacer esto con el comando `print`:

```
class Hola {  
  
    // main: genera una salida simple  
  
    public static void main (String[] args) {  
        System.out.print ("¡Adiós, ");  
        System.out.println ("mundo cruel!");  
    }  
}
```

En este caso la salida en una sola línea como `Adiós, mundo cruel!`. Nótese que hay un espacio entre la palabra “Adiós” y las segundas comillas. Este espacio aparece en la salida, por lo tanto afecta el comportamiento del programa. Aquellos espacios que aparecen fuera de las comillas no afectan el comportamiento del programa. Por ejemplo, yo podría haber escrito:

```
class Hola {  
    public static void main (String[] args) {  
        System.out.print ("¡Adiós, ");  
        System.out.println ("mundo cruel!");  
    }  
}
```

Este programa compilaría y correría tan bien como el original. Los saltos de línea (*newlines*) tampoco afectan al comportamiento del programa, de modo que podría haber escrito:

```
class Hola { public static void main (String[] args) {  
    System.out.print ("¡Adiós, "); System.out.println  
    ("mundo cruel!");}}
```

Eso funcionaría también, aunque probablemente habrás podido notar que el programa se va tornando cada vez más y más difícil de leer. Los espacios y los saltos de línea son útiles para organizar tu programa visualmente, haciéndolo más fácil de leer y haciendo también más sencillo encontrar errores de sintaxis.

2.2 Variables

Una de las características más poderosas de un lenguaje de programación es su capacidad para manipular **variables**. Una variable es el nom-

bre de una ubicación donde se almacena un **valor**. Los valores son cosas que se pueden imprimir y almacenar y (como veremos más adelante) con las que es posible operar. Las cadenas que hemos estado imprimiendo ("Hola, Mundo.", "Adiós, ", etc.) son valores.

Si querés almacenar un valor, debés crear una variable. Dado que los valores que queremos almacenar son cadenas, vamos a declarar que esa nueva variable sea una cadena.

```
String pedro;
```

Esta sentencia es una **declaración**, porque declara que la variable llamada pedro tiene tipo String (cadena). Cada variable tiene un tipo que determina justamente qué tipo de valor es capaz de almacenar. Por ejemplo, el tipo int puede almacenar enteros, y no será sorprendente que el tipo String puede almacenar cadenas.

Habrás notado que algunos tipos comienzan con mayúscula y otros con minúscula. Veremos cuál es la importancia de esta distinción más adelante, pero por ahora deberías tenerlo bien claro. No existen tipos como Int o string, y el compilador se quejará si inventás uno.

Para crear una variable de tipo entero, la sintaxis es `int bruno;`, donde bruno es el nombre arbitrario que decidimos usar para la variable. En general, querrás nombrar a las variables de modo que su nombre indique lo que planeás hacer con ellas. Por ejemplo, si vieras estas declaraciones de variables:

```
String nombre;  
String apellido;  
int hora, minuto;
```

probablemente puedas adivinar correctamente qué valores se guardarán en ellas. Este ejemplo también muestra la sintaxis para declarar múltiples variables con el mismo tipo: hora y minuto son ambas enteras (de tipo int).

2.3 Asignación

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

```
pepe = "Hola.";      // le da a pepe el valor "Hola."  
hora = 11;           // le asigna el valor 11 a hora  
minuto = 59;         // deja minuto en 59
```

Este ejemplo muestra tres asignaciones, y los comentarios muestran tres maneras diferentes en que la gente se refiere a las sentencias de asignación. El vocabulario puede ser confuso aquí, pero la idea es simple:

- Cuando declaras una variable, estás creando una ubicación de almacenamiento con un nombre.
- Cuando asignas una variable, le estás dando un valor.

Una manera común de representar variables en papel es dibujando una caja con el nombre de la variable afuera y el valor de la variable adentro. La figura muestra el efecto de tres sentencias de asignación:

pepe	"Hola."
hora	11
minuto	59

Para cada variable, el nombre de la variable aparece afuera de la caja y el valor aparece adentro.

Como regla general, una variable debe tener el mismo tipo que el valor que se le asigna. No se puede almacenar una cadena en `minuto` o un entero en `pepe`. Por otro lado, esta regla puede confundir, porque hay muchas formas de convertir valores de un tipo a otro, y Java en algunas ocasiones convierte cosas automáticamente. Así que por ahora es suficiente que recuerdes la regla general, y volveremos a hablar de los casos especiales más adelante.

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

```
pepe = "123";    // válido
pepe = 123;      // inválido
```

2.4 Imprimiendo variables

Se puede imprimir el valor de una variable utilizando los mismos comandos que usamos para imprimir `Strings` (cadenas).

```

class Hola {
    public static void main (String[] args) {
        String primeraLinea;
        primeraLinea = "Hola, otra vez!";
        System.out.println (primeraLinea);
    }
}

```

Este programa crea una variable llamada `primeraLinea`, le asigna el valor "Hola, otra vez!" y luego imprime ese valor. Cuando hablamos de "imprimir una variable", nos referimos a imprimir el *valor* de la variable. Para imprimir el *nombre* de la variable, debés ponerlo entre comillas. Por ejemplo: `System.out.println ("primeraLinea");`. Haciéndolo un poquito más difícil, podrías escribir

```

String primeraLinea;
primeraLinea = "Hola, otra vez!";
System.out.print ("El valor de primeraLinea es ");
System.out.println (primeraLinea);

```

La salida de este programa es

El valor de `primeraLinea` es Hola, otra vez!

Estoy encantado de informar que la sintaxis para imprimir una variable es idéntica sin importar el tipo de la misma.

```

int hora, minuto;
hora = 11;
minuto = 59;
System.out.print ("La hora actual es ");
System.out.print (hora);
System.out.print (":");
System.out.print (minuto);
System.out.println (".");

```

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

ADVERTENCIA: Es una práctica común usar varios comandos `print` seguidos de un `println`, con el fin de imprimir múltiples valores en la misma línea. Pero se debe ser cuidadoso para recordar el `println` al final. En muchos entornos, la salida de `print` es almacenada sin ser mostrada hasta que se llama al comando `println`, punto en el cual se muestra toda la línea de inmediato. Si se omite el `println`, ¡el programa puede llegar a terminar sin jamás mostrar la salida almacenada!

2.5 Palabras reservadas

Algunas secciones atrás, dije que se podía inventar cualquier nombre a antojo para las variables, pero eso no es completamente cierto. Hay ciertas palabras que son reservadas en Java porque son usadas por el compilador para analizar sintácticamente la estructura de tu programa, y si se las usa como nombres de variables, se confundirá. Estas palabras, llamadas **palabras reservadas**², incluyen `void`, `class`, `public`, `int`, y muchas más. La lista completa está disponible en http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html. Este libro, provisto por Sun, incluye documentación Java a la que iré haciendo referencia a lo largo del libro.

Más que memorizar la lista, sugiero que aproveches una característica que proveen muchos entornos de desarrollo Java: el coloreado de código. A medida que uno escribe, diferentes partes del programa van apareciendo en diferentes colores. Por ejemplo, las palabras reservadas pueden ser azules, las cadenas rojas, y el resto del código negro. Si uno escribe un nombre de variable y se pone azul, ¡cuidado! Puede ser que el compilador comience a comportarse de manera extraña.

2.6 Operadores

Los **operadores** son símbolos especiales que se usan para representar cálculos simples como la suma y la multiplicación. La mayoría de los operadores en Java hacen exactamente lo que uno se imagina, porque son símbolos comunes de la matemática. Por ejemplo, el operador para sumar dos enteros es `+`. Las siguientes son expresiones Java cuyo significado es bastante obvio:

`1+1` `hora-1` `hora*60 + minuto` `minuto/60`

Las expresiones pueden contener tanto nombres de variables como números. En cada caso el nombre de la variable es reemplazado por su valor antes de que se realice el cálculo.

La suma, la resta y la multiplicación, todas hacen lo que uno espera, pero puede que te sorprenda con la división. Por ejemplo, el siguiente programa:

2. N.d.T: En inglés se llaman “keywords”, que literalmente significaría “palabras clave”.

```

int hora, minuto;
hora = 11;
minuto = 59;
System.out.print ("Cantidad de minutos desde la medianoche: ");
System.out.println (hora*60 + minuto);
System.out.print ("Fraccion de hora que ha transcurrido: ");
System.out.println (minuto/60);

```

generaría la siguiente salida:

```

Cantidad de minutos desde la medianoche: 719
Fraccion de hora que ha transcurrido: 0

```

La primera línea es tal cual la esperábamos, pero la segunda es rara. El valor de la variable `minuto` es 59, y 59 dividido 60 es 0.98333, no 0. La discrepancia se debe a que Java realiza una **división entera**.

Cuando ambos de los **operandos** son enteros (los operandos son las cosas sobre las cuales se opera), el resultado debe ser también entero, y por convención, la división entera siempre redondea el resultado hacia *abajo*, incluso en casos como este en los que el siguiente entero está tan cerca. Una posible alternativa en este caso es calcular un porcentaje, en lugar de una fracción:

```

System.out.print ("Porcentaje de la hora que ha transcurrido: ");
System.out.println (minuto*100/60);

```

El resultado es:

```

Porcentaje de la hora que ha transcurrido: 98

```

Una vez más, el resultado es redondeado hacia abajo, pero al menos ahora la respuesta es aproximadamente correcta. A fin de obtener una respuesta aún más precisa, podríamos usar un tipo diferente de variable, llamada variable de punto-flotante, que es capaz de almacenar valores fraccionarios. Veremos eso en el siguiente capítulo.

2.7 Orden de las operaciones

Cuando en una expresión aparecen más de un operador, el orden de evaluación depende de las reglas de **precedencia**. Una explicación completa de precedencia puede tornarse complicada, pero para empezar:

- La multiplicación y la división tienen precedencia sobre (suceden antes que) la suma y la resta. Entonces $2*3-1$ da 5, no 4, y $2/3-1$ da -1, no 1 (no olvides que en la división entera $2/3$ es 0).

- Si los operadores tienen la misma precedencia son evaluados de izquierda a derecha. Entonces en la expresión `minuto*100/60`, la multiplicación sucede primero, dando como resultado `5900/60`, que a su vez da 98. Si la operación fuera de derecha a izquierda, el resultado sería `59*1` que es 59, que es erróneo.
- Cada vez que quieras hacer caso omiso de las reglas de precedencia (o no estés seguro de cuáles son) podés usar paréntesis. Las expresiones entre paréntesis son evaluadas primero, entonces `2*(3-1)` es 4. También podés usar paréntesis con el fin de que las expresiones sean más fácilmente legibles, como en `(minuto * 100) / 60`, aunque el resultado sea el mismo.

2.8 Operadores de cadenas

En general no se pueden realizar operaciones matemáticas en Strings (cadenas en Java), incluso cuando las cadenas parezcan números. Las siguientes expresiones son incorrectas (asumiendo que `pepe` tiene tipo String)

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

Dicho sea de paso, ¿Se puede saber con sólo mirar esas expresiones si `pepe` es un entero o una cadena? No. La única manera de saber el tipo de una variable es mirando el lugar en el que fue declarada.

Interesantemente, el operador `+` sí funciona con Strings, aunque no hace exactamente lo que esperarías. Para Strings, el operador `+` representa la **concatenación**, que es simplemente enlazar los operandos uno detrás del otro. Entonces, `"Hola, " + "mundo."` da como resultado la cadena `"Hola, mundo."` y `pepe + "ismo"` agrega el sufijo *ismo* al final de lo que sea que contenga la variable `pepe`, lo cual es una manera práctica de darle nombres a nuevas formas de intolerancia.

2.9 Composición

Hasta ahora hemos hecho un repaso por los elementos de un lenguaje de programación—variables, expresiones y sentencias—aisladamente, sin hablar de cómo combinarlos.

Una de las características más útiles de los lenguajes de programación es que permiten tomar pequeños bloques de construcción y **componerlos**. Por ejemplo, sabemos cómo multiplicar números y también sabemos cómo imprimir; sucede que podemos hacer ambas cosas al mismo tiempo:

```
System.out.println (17 * 3);
```

En realidad, no debería decir “al mismo tiempo”, dado que la verdad es que la multiplicación tiene que ocurrir antes que la impresión (el print), pero el punto es que cualquier expresión, que tenga que ver con números, cadenas, y/o variables, puede ser usada dentro de una sentencia de impresión (print). Ya hemos visto un ejemplo:

```
System.out.println (hora*60 + minuto);
```

Pero también se pueden poner expresiones arbitrarias en el lado derecho de una sentencia de asignación:

```
int porcentaje;  
porcentaje = (minuto * 100) / 60;
```

Esta habilidad puede no parecer tan impresionante ahora mismo, pero veremos otros ejemplos en los que la composición hace posible expresar cálculos complicados de manera prolija y concisa.

ADVERTENCIA: Existen límites para los lugares en los que se pueden usar ciertas expresiones; más que nada, los lados izquierdos de una sentencia de asignación tienen que ser nombres de *variables*, no expresiones. Esto es así porque el lado izquierdo indica la ubicación en donde se almacenará el resultado. Las expresiones no representan ubicaciones de almacenamiento, sólo valores. Por lo tanto, lo siguiente es incorrecto: `minuto+1 = hora;`.

2.10 Glosario

variable: Ubicación de almacenamiento con nombre. Todas las variables tienen un tipo, el cual es declarado en el momento de su creación.

valor: Número o cadena (u otra cosa que será mencionada más adelante) que puede ser almacenado en una variable. Cada valor pertenece a un tipo.

tipo: Conjunto de valores. El tipo de una variable determina qué valores pueden ser almacenados en ella. Hasta el momento, los tipos que hemos visto son enteros (`int` en Java) y cadenas (`String` en Java).

palabra reservada: Palabra usada por el compilador para analizar sintácticamente un programa. No se pueden utilizar palabras reservadas, tales como `public`, `class` y `void` como nombres de variables.

sentencia: Una línea de código que representa un comando o acción. Hasta el momento, las sentencias que hemos visto son declaraciones, asignaciones, y sentencias de impresión (print).

declaración: Sentencia que crea una variable nueva y determina su tipo.

asignación: Sentencia que asigna un valor a una variable.

expresión: Combinación de variables, operadores y valores que representan un solo valor. Las expresiones además tienen tipos, determinados por los operadores y los operandos.

operador: Símbolo especial que representa un cálculo simple como suma, multiplicación o concatenación de cadenas.

operando: Uno de los valores con los que opera un operador.

precedencia: El orden en que son evaluadas las operaciones.

concatenar: Enlazar dos operandos uno detrás del otro.

composición: La habilidad de combinar expresiones simples y sentencias, en sentencias y expresiones compuestas a fin de representar cálculos complejos de manera concisa.

2.11 Ejercicios

Ejercicio 2.1

- Creá un nuevo programa llamado Fecha.java. Copiá o tipeá algo como el programa “Hola, Mundo” y asegurate de poder compilarlo y correrlo.
- Siguiendo el ejemplo en la Sección 2.4, escribí un programa que cree variables llamadas diaDeLaSemana, diaDelMes, mes y anio. dia contendrá el día de la semana y diaDelMes contendrá el día del mes. ¿De qué tipo es cada variable? Asigná valores a esas variables para que representen la fecha de hoy.
- Imprimí el valor de cada variable en una línea distinta. Esto es un paso intermedio que sirve para chequear que todo está funcionando bien de momento.
- Modificá el programa para que imprima la fecha en el estándar argentino como: Lunes 9 de agosto de 2010 .
- Modificá el programa de nuevo para que ahora la salida total sea:

Formato argentino:
Lunes 9 de agosto de 2010
Formato americano:
Lunes, agosto 9, 2010
Formato europeo:
Lunes 9 agosto, 2010

El objetivo de este ejercicio es usar concatenación de cadenas para mostrar valores de diferentes tipos (`int` y `String`), y practicar el desarrollo gradual de programas añadiendo sentencias de a unas pocas por vez.

Ejercicio 2.2

- Creá un nuevo programa llamado `Hora.java`. De ahora en adelante no te recordaré que comiences con un pequeño programa ya funcionando pero deberías hacerlo.
- Siguiendo el ejemplo en la Sección 2.6, creá variables llamadas `hora`, `minuto` y `segundo`, y asignales valores que sean más o menos la hora actual. Usá un reloj de 24 horas, de modo que a las 2pm el valor de `hora` sea 14.
- Hacé que el programa calcule e imprima el número de segundos desde la medianoche.
- Hacé que el programa calcule e imprima el número de segundos que le quedan al día.
- Hacé que el programa calcule e imprima el porcentaje del día que transcurrió.
- Cambiá los valores de `hora`, `minuto` y `segundo` para que reflejen la hora actual (asumo que transcurrió algo de tiempo), y chequeá que el programa funcione correctamente con diferentes valores.

El punto de este ejercicio es usar algunas de las operaciones aritméticas, y empezar a pensar acerca de entidades compuestas como el tiempo del día, que son representadas con múltiples valores. Además, podés llegar a encontrarte con problemas calculando porcentajes con enteros, que es la motivación para los números de punto flotante en el siguiente capítulo.

PISTA: tal vez quieras usar variables adicionales para retener valores temporalmente durante los cálculos. A variables como esta, que son usadas en los cálculos pero nunca impresas, se las suele llamar variables temporales o intermedias.

Capítulo 3

Métodos

3.1 Punto flotante

En el último capítulo tuvimos algunos problemas al tratar con números que no eran enteros. En ese caso salvamos el problema midiendo porcentajes en vez de fracciones, pero una solución más general es usar números de punto flotante, los cuales pueden representar tanto fracciones como enteros. En Java, el tipo de los números de punto flotante se llama `double`¹. Es posible crear variables de punto flotante y asignarles valores usando la misma sintaxis que usamos para los otros tipos. Por ejemplo:

```
double pi;  
pi = 3.14159;
```

También es válido declarar una variable y asignarle un valor a la misma, al mismo tiempo:

```
int x = 1;  
String vacia = "";  
double pi = 3.14159;
```

De hecho, esta sintaxis es muy común. Una declaración combinada con una asignación es a veces llamada una **inicialización**.

Si bien los números de punto flotante son útiles, a veces son una fuente de confusión ya que parece haber una superposición entre números enteros y de punto flotante. Por ejemplo, si tenemos el valor 1, ¿es un entero, un número de punto flotante o ambos?

1. N.d.T: Doble precisión, en relación al tipo `float`.

Estrictamente hablando, Java distingue el valor entero 1 del valor de punto flotante 1.0, incluso aunque ambos parezcan ser el mismo número. Ellos pertenecen a distintos tipos y, estrictamente hablando, no está permitido hacer asignaciones entre distintos tipos. Por ejemplo, lo siguiente es inválido,

```
int x = 1.1;
```

debido a que la variable de la izquierda es un `int` (de entero) y el valor de la derecha es un `double`. Pero es fácil olvidar esta regla, especialmente porque hay lugares donde Java convertirá automáticamente de un tipo a otro. Por ejemplo:

```
double y = 1;
```

técnicamente no debería ser válido, pero Java lo permite convirtiendo el `int` a un `double` automáticamente. Esta facilidad es conveniente, pero puede causar problemas, por ejemplo:

```
double y = 1 / 3;
```

Podría esperarse que a la variable `y` le sea asignado el valor 0.333333, que es un valor de punto flotante válido, pero sin embargo obtendrá el valor 0.0. La razón es que la expresión de la derecha parece ser el cociente entre dos enteros, por lo que Java realiza la división *entera*, la cual da el valor 0. Convertida a punto flotante, el resultado es 0.0.

Una forma de solucionar este problema (una vez que lo encuentres) es hacer el lado derecho una expresión de punto flotante:

```
double y = 1.0 / 3.0;
```

Esto establece el valor de `y` en 0.333333, como era de esperar.

Todas las operaciones que hemos visto hasta ahora —suma, resta, multiplicación y división— también funcionan en valores de punto flotante, aunque es interesante saber que el mecanismo subyacente es completamente diferente. De hecho, la mayoría de los procesadores poseen hardware especial sólo para realizar operaciones de punto flotante.

3.2 Convirtiendo entre `double` e `int`

Como se mencionó, Java convierte a los `ints` en `doubles` automáticamente si es necesario, porque ninguna información se pierde en la conversión. Por otro lado, ir desde un `double` a un `int` requiere redondeo. Java no realiza esta operación automáticamente, de manera de garantizar que

como programador estés al tanto de la pérdida de la parte fraccional del número. La forma más simple de convertir un valor de punto flotante a entero es usar un **casteo de tipos**². El casteo de tipos se llama de esa manera porque permite tomar un valor que pertenece a un tipo y “convertirlo” a otro tipo (en el sentido de amoldarlo o reformarlo, no tirarlo).

Desafortunadamente, la sintaxis para el casteo de tipos es fea: se pone el nombre del tipo entre paréntesis y se usa como un operador. Por ejemplo:

```
int x = (int) Math.PI;
```

El operador `(int)` tiene el efecto de convertir lo que sigue en un entero, por lo que `x` obtiene el valor 3.

El casteo de tipos tiene precedencia sobre las operaciones aritméticas, por lo que en el siguiente ejemplo, el valor de `PI` se convierte a un entero primero, y el resultado es 60, no 62.

```
int x = (int) Math.PI * 20.0;
```

La conversión a entero siempre redondea hacia abajo, incluso si la parte fraccionaria es 0.99999999.

Estas dos propiedades (precedencia y redondeo) pueden hacer que el casteo de tipos sea poco elegante.

3.3 Métodos de la clase Math

En matemática, probablemente hayas visto funciones como \sin y \log , y hayas aprendido a evaluar expresiones como $\sin(\pi/2)$ y $\log(1/x)$. Primero, se evalúa la expresión en paréntesis, la cual se llama **argumento** de la función. Por ejemplo, $\pi/2$ es aproximadamente 1.571, y $1/x$ es 0.1 (asumiendo que x es 10).

Entonces, es posible evaluar la función misma, ya sea mirando en una tabla o realizando varios cálculos. El \sin de 1.571 es 1, y el \log de 0.1 es -1 (asumiendo que \log indica el logaritmo en base 10).

Este proceso puede ser aplicado repetidas veces para evaluar expresiones más complicadas como $\log(1/\sin(\pi/2))$. Primero evaluamos el argumento de la función de más adentro, después evaluamos la función, y así seguimos.

Java provee un conjunto de funciones preincorporadas que incluye la mayoría de las operaciones matemáticas que puedas imaginar. Estas funciones son llamadas **métodos**. La mayoría de los métodos matemáticos operan con doubles.

2. También llamado conversión de tipos.

Los métodos matemáticos son llamados usando una sintaxis que es similar a la de los comandos `print` que ya hemos visto:

```
double raiz = Math.sqrt (17.0);
double angulo = 1.5;
double altura = Math.sin (angulo);
```

El primer ejemplo asigna a `raiz` la raíz cuadrada de 17. El segundo ejemplo encuentra el seno de 1.5, el cual es el valor de la variable `angulo`. Java asume que los valores que usás con `sin` y las otras funciones trigonométricas (`cos`, `tan`) están en *radianes*. Una forma de convertir de grados a radianes, es dividir por 360 y multiplicar por 2π . Convenientemente, Java provee a π como valor preincorporado:

```
double grados = 90;
double angulo = grados * 2 * Math.PI / 360.0;
```

Notar que `PI` está todo en letras mayúsculas. Java no reconoce `Pi`, `pi`, ni `pie`.

Otro método útil en la clase `Math` es `round` (redondear), el cual redondea un valor de punto flotante al entero más cercano y devuelve un `int`.

```
int x = Math.round (Math.PI * 20.0);
```

En este caso la multiplicación sucede primero, antes de que el método sea llamado. El resultado es 63 (redondeado hacia arriba desde 62.8319).

3.4 Composición

Tal como con las funciones matemáticas, los métodos en Java pueden ser **compuestos**, lo cual significa que es posible usar una expresión como parte de otra. Por ejemplo, es posible usar cualquier expresión como argumento de un método:

```
double x = Math.cos (angulo + Math.PI/2);
```

Esta sentencia toma el valor `Math.PI`, lo divide por dos y suma el resultado al valor de la variable `angulo`. La suma es entonces pasada como un argumento al método `cos`. (Notar que `PI` es el nombre de una variable, no un método, por lo que no hay argumentos, ni siquiera el argumento vacío `()`).

También podemos tomar el resultado de un método y pasarlo como argumento de otro:

```
double x = Math.exp (Math.log (10.0));
```

En Java, la función `log` siempre usa base e , por lo que esta sentencia encuentra el logaritmo en base e de 10 y después eleva e a esa potencia. El resultado es asignado a `x`; espero que sepas cuál es.

3.5 Agregando nuevos métodos

Hasta ahora, sólo hemos estado usando los métodos que vienen preincorporados en Java, pero también es posible agregar nuevos métodos. En realidad, ya hemos visto una definición de un método: `main`³. El método llamado `main` es especial porque indica dónde empieza la ejecución de un programa, pero la sintaxis de `main` es la misma que para cualquier definición de métodos:

```
public static void NOMBRE ( LISTA DE PARAMETROS ) {  
    SENTENCIAS  
}
```

Podés inventar cualquier nombre que quieras para tu método, excepto que no podés llamarlo `main` o alguna otra palabra reservada de Java. La lista de parámetros especifica qué información hay que proveer, si es que la hay, para poder usar (o **llamar**) la nueva función.

El único parámetro de `main` es `String[] args`, que indica que quienquiera que invoque a `main` tiene que proveer un arreglo de Strings (nos detendremos en los arreglos en el Capítulo 10). Los primeros dos métodos que vamos a escribir no tienen parámetros, por lo que la sintaxis se ve así:

```
public static void nuevaLinea () {  
    System.out.println ("");  
}
```

Este método se llama `nuevaLinea`, y los paréntesis vacíos indican que no toma parámetros. Contiene solamente una única sentencia, que imprime una cadena vacía, indicada por `""`. Imprimir un valor de tipo `String` sin ninguna letra puede no parecer útil, pero hay que recordar que `println` saltea a la siguiente línea después de imprimir, por lo que esta sentencia tiene el efecto de saltar a la línea siguiente.

En `main` podemos llamar a este nuevo método usando una sintaxis similar a la forma en que llamamos a los comandos preincorporados de Java:

3. N.d.T: “principal” en inglés.


```

public static void main (String[] args) {
    System.out.println ("Primera linea.");
    nuevaLinea ();
    System.out.println ("Segunda linea.");
}

```

La salida del programa es

Primera linea.

Segunda linea.

Notar el espacio extra entre las dos líneas. ¿Qué tal si quisiéramos más espacio entre las líneas? Podríamos llamar al mismo método repetidamente:

```

public static void main (String[] args) {
    System.out.println ("Primera linea.");
    nuevaLinea ();
    nuevaLinea ();
    nuevaLinea ();
    System.out.println ("Segunda linea.");
}

```

O podríamos escribir un nuevo método, llamado `tresLineas`, que imprima tres nuevas líneas:

```

public static void tresLineas () {
    nuevaLinea (); nuevaLinea (); nuevaLinea ();
}

public static void main (String[] args) {
    System.out.println ("Primera linea.");
    tresLineas ();
    System.out.println ("Segunda linea.");
}

```

Deberías notar algunas cosas sobre este programa:

- Se puede llamar al mismo procedimiento repetidamente. De hecho, es muy útil hacer eso.
- Se puede hacer que un método invoque a otro método. En este caso, `main` llama a `tresLineas` y `tresLineas` llama a `nuevaLinea`. Otra vez, esto es común y muy útil.

- En `tresLineas` escribimos tres sentencias en la misma línea de código, lo cual es válido sintácticamente (recordar que los espacios y las nuevas líneas usualmente no cambian el significado de un programa). Por otro lado, es usualmente una mejor idea poner cada sentencia en una línea distinta, para hacer tu programa más fácil de leer. A veces rompemos esa regla en este libro para ahorrar espacio.

Hasta ahora, puede no estar claro por qué vale la pena crear todos estos nuevos métodos. En realidad, hay un montón de razones, pero este ejemplo demuestra solamente dos:

1. Crear un nuevo método brinda una oportunidad de dar un nombre a un grupo de sentencias. Los métodos pueden simplificar un programa al esconder un cómputo complejo detrás de un comando simple, y usando una frase en castellano en lugar de código complicado. ¿Qué es más claro, `nuevaLinea` o `System.out.println (")"`?
2. Crear un nuevo método puede hacer un programa más corto al eliminar código repetitivo. Por ejemplo, ¿cómo harías para imprimir por pantalla nueve líneas nuevas consecutivas? Podrías simplemente llamar `tresLineas` tres veces.

3.6 Clases y métodos

Reuniendo todos los fragmentos de código de la sección anterior, la definición completa de la clase se ve así:

```
class NuevaLinea {  
  
    public static void nuevaLinea () {  
        System.out.println (");  
    }  
  
    public static void tresLineas () {  
        nuevaLinea (); nuevaLinea (); nuevaLinea ();  
    }  
  
    public static void main (String[] args) {  
        System.out.println ("Primera linea.");  
        tresLineas ();  
        System.out.println ("Segunda linea.");  
    }  
}
```

La primera línea indica que esta es la definición de una nueva clase llamada NuevaLinea. Una clase es una colección de métodos relacionados. En este caso, la clase llamada NuevaLinea contiene tres métodos, llamados nuevaLinea, tresLineas, y main.

La otra clase que hemos visto es la clase Math. Ella contiene métodos llamados sqrt, sin, y muchos otros. Cuando llamamos una función matemática, tenemos que especificar el nombre de la clase (Math) y el nombre de la función. Es por eso que la sintaxis es ligeramente diferente entre los métodos preincorporados y los métodos que escribimos nosotros:

```
Math.pow (2.0, 10.0);  
nuevaLinea ();
```

La primera sentencia llama al método pow de la clase Math class (que eleva el primer argumento a la potencia del segundo argumento)⁴. La segunda sentencia llama el método nuevaLinea, que Java asume (correctamente) está en la clase NuevaLinea, que es la que estamos escribiendo.

Si tratás de llamar un método de una clase errónea, el compilador va a generar un error. Por ejemplo, si tipearas:

```
pow (2.0, 10.0);
```

El compilador va a decir algo como “No puedo encontrar un método llamado pow en la clase NuevaLinea.” Si has visto este mensaje, podrías haberte preguntado por qué estaba buscando pow en tu definición de clase. Ahora ya lo sabés.

3.7 Programas con múltiples métodos

Cuando uno mira una definición de clase que contiene varios métodos, es tentador leerla de arriba hacia abajo, pero eso es probable que resulte confuso, ya que ese no es el **orden de ejecución** del programa.

La ejecución siempre comienza en la primera sentencia de main, independientemente de dónde esté en el programa (en este caso lo puse deliberadamente al final). Las sentencias son ejecutadas una a la vez, en orden, hasta alcanzar una llamada de un método. Las llamadas de métodos son como una desviación en el flujo de ejecución. En vez de ir a la siguiente sentencia, se va a la primera línea del método llamado, ejecutando todas las sentencias ahí, y después se vuelve y retoma donde se había dejado.

Esto suena bastante simple, excepto que hay que recordar que un método puede llamar otro. De esta manera, mientras estamos en la mitad

4. N.d.T.: pow viene de power, que significa potencia.

de `main`, podríamos tener que salir y ejecutar sentencias en `tresLineas`. Pero mientras estamos ejecutando `tresLineas`, somos interrumpidos tres veces y vamos a ejecutar `nuevaLinea`.

Por su parte, `nuevaLinea` llama al método preincorporado `println`, que causa a su vez otra desviación. Afortunadamente, Java es un experto en llevar la cuenta de dónde está, por lo que cuando `println` termine, retoma por donde había dejado en `nuevaLinea`, luego retrocede hasta `tresLineas`, y por último vuelve a `main` de manera que el programa pueda terminar.

En realidad, técnicamente, el programa no termina al final de `main`. En cambio, la ejecución retoma donde había dejado en el programa que invocó a `main`, el cual es el intérprete de Java. El intérprete de Java se ocupa de cosas como la eliminación de ventanas y hacer una limpieza general, y luego el programa termina.

¿Cuál es la moraleja de esta sórdida historia? Cuando leas un programa, no lo leas desde arriba hacia abajo. En vez de eso, seguí el flujo de ejecución.

3.8 Parámetros y argumentos

Algunos de los métodos preincorporados que hemos usado tienen **parámetros**, que son valores que se le proveen para que puedan hacer su trabajo. Por ejemplo, si queremos encontrar el seno de un número, tenemos que indicar qué número es. Por ello, sin toma un valor `double` como parámetro. Para imprimir una cadena, hay que proveer la cadena, y es por eso que `println` toma un `String` como parámetro.

Algunos métodos toman más de un parámetro, como `pow`, el cual toma dos `doubles`, la base y el exponente.

Notar que en cada uno de esos casos tenemos que especificar no solo cuántos parámetros hay, sino también de qué tipo son. Por eso no debería sorprender que cuando escribimos una definición de clase, la lista de parámetros indica el tipo de cada parámetro. Por ejemplo:

```
public static void imprimirDosVeces (String rigoberto) {  
    System.out.println (rigoberto);  
    System.out.println (rigoberto);  
}
```

Este método toma un solo parámetro, llamado `rigoberto`, que tiene tipo `String`. Cualquiera sea ese parámetro (y en este punto no tenemos idea de cuál es), es impreso dos veces. Yo elegí el nombre `rigoberto` para sugerir que el nombre que das como parámetro depende de vos, pero en general querrías elegir algo más ilustrativo que `rigoberto`.

Para llamar este método, tenemos que proveer un String. Por ejemplo, podríamos tener un método main como este:

```
public static void main (String[] args) {  
    imprimirDosVeces ("No me hagas decir esto dos veces!");  
}
```

La cadena que proporcionaste se denomina **argumento**, y decimos que el argumento es **pasado** al método. En este caso, estamos creando una cadena que contiene el texto “No me hagas decir esto dos veces!” y pasando esa cadena como un argumento a `imprimirDosVeces` donde, contrario a sus deseos, será impreso dos veces.

Alternativamente, si tuviéramos una variable de tipo String podríamos usarla como un argumento en vez de lo anterior:

```
public static void main (String[] args) {  
    String argumento = "Nunca digas nunca.";   
    imprimirDosVeces (argumento);  
}
```

Notar algo muy importante aquí: el nombre de la variable que pasamos como argumento (`argumento`) no tiene nada que ver con el nombre del parámetro (`rigoberto`). Permite decirlo nuevamente:

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

Pueden ser el mismo o pueden ser diferentes, pero es importante darse cuenta de que no son la misma cosa, excepto que sucede que tienen el mismo valor (en este caso la cadena “Nunca digas nunca.”).

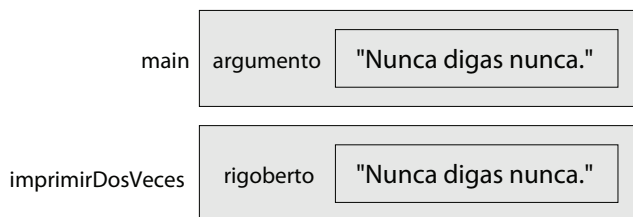
El valor que proveas como argumento debe tener el mismo tipo que el parámetro del método que invocás. Esta regla es muy importante, pero a menudo se complica en Java por dos razones:

- Hay algunos métodos que pueden aceptar argumentos con varios tipos diferentes. Por ejemplo, es posible mandar *cualquier* tipo a `print` y `println`, y hará lo correcto sin importar qué. Este tipo de cosas es una excepción, de todas maneras.
- Si violás esta regla, el compilador suele generar un mensaje de error confuso. En vez de decir algo como “Estás pasando un argumento de tipo erróneo a este método,” probablemente diga algo del estilo de que no pudo encontrar un método con ese nombre que acepte un argumento de ese tipo. Una vez que hayas visto este mensaje de error un par de veces, sin embargo, vas a darte cuenta de cómo interpretarlo.

3.9 Diagramas de la pila de ejecución

Los parámetros y otras variables sólo existen dentro de sus propios métodos. En los confines del `main`, no existe tal cosa como `rigoberto`. Si tratás de usarlo, el compilador se quejará. De forma similar, dentro de `imprimirDosVeces` no existe tal cosa como argumento.

Una forma de llevar la cuenta de dónde cada variable está definida es con un **diagrama de la pila de ejecución**. El diagrama de la pila para el ejemplo anterior se ve así:



Para cada método hay un recuadro gris llamado **frame**⁵ que contiene los parámetros de los métodos y las variables locales. El nombre del método aparece afuera del frame. Como de costumbre, el valor de cada variable es dibujado dentro de un recuadro con el nombre de la variable al lado de él.

3.10 Métodos con múltiples parámetros

La sintaxis para declarar y llamar métodos con múltiples parámetros es una fuente común de errores. Primero, recordemos que hay que declarar el tipo de cada parámetro. Por ejemplo

```
public static void imprimirTiempo(int hora, int minuto) {
    System.out.print (hora);
    System.out.print (":");
    System.out.println (minuto);
}
```

Puede parecer tentador escribir `int hora, minuto`, pero ese formato sólo es válido para declaraciones de variables, no para parámetros.

Otra fuente común de confusión es que no hay que declarar los tipos de los argumentos al llamar un método. Lo siguiente es erróneo!

5. N.d.T.: *frame* significa *marco*.

```
int hora = 11;
int minuto = 59;
imprimirTiempo (int hora, int minuto);    // ERROR!
```

En este caso, Java ya sabe el tipo de hora y minuto porque ha visto sus declaraciones. Es innecesario e inválido incluir el tipo cuando se pasa como argumento. La sintaxis correcta es `imprimirTiempo (hora, minuto)`.

Ejercicio 3.1

Dibujar un diagrama de la pila que muestre el estado del programa cuando `main` invoque a `imprimirTiempo` con los argumentos 11 y 59.

3.11 Métodos con resultados

Quizás hayas notado a esta altura que algunos de los métodos que estamos usando, tales como los métodos `Math`, producen resultados. Otros métodos, como `println` y `nuevaLinea`, realizan alguna acción pero no devuelven ningún valor. Esto plantea algunas preguntas:

- ¿Qué pasa si llamamos a un método y no queremos hacer nada con el resultado (por ej. no vamos a asignarlo a ninguna variable ni usarlo como parte de una expresión más grande)?
- ¿Qué pasa si usamos un método `print` como parte de una expresión, tal como `System.out.println ("buu!") + 7`?
- ¿Podemos escribir métodos que produzcan resultados, o estamos atascados con cosas como `nuevaLinea` e `imprimirDosVeces`?

La respuesta a la tercera pregunta es “sí, es posible escribir métodos que devuelvan valores”, y vamos a hacerlo en un par de capítulos. Te dejo a vos que respondas las otras dos preguntas intentando ver qué pasa. De hecho, siempre que tengas una pregunta acerca de qué es válido o inválido en Java, una buena forma de averiguarlo es preguntarle al compilador.

3.12 Glosario

punto flotante: Tipo de variable (o valor) que puede contener tanto fracciones como enteros. En Java este tipo se llama `double`.

clase: Nombre para una colección de métodos. Hasta ahora, hemos usado las clases `Math` y `System` y hemos escrito clases llamadas `Hola` y `NuevaLinea`.

método: Nombre para una secuencia de sentencias que realiza alguna función útil. Los métodos pueden o no tomar parámetros, y pueden o no producir un resultado.

parámetro: Pieza de información que se provee al llamar un método. Los parámetros son como las variables en el sentido de que contienen valores y son de algún tipo.

argumento: Valor que se provee cuando se llama a un método. Este valor debe tener el mismo tipo que el parámetro correspondiente. En castellano, parámetro y argumento tienden a utilizarse indistintamente.

llamar: Causar que un método sea ejecutado.

3.13 Ejercicios

Ejercicio 3.2

El objetivo de este ejercicio es practicar la lectura de código y asegurate que entiendas el flujo de ejecución de un programa con múltiples métodos.

- a. ¿Cuál es la salida del siguiente programa? Sé preciso acerca de dónde hay espacios y dónde hay nuevas líneas.

AYUDA: Empezá por describir en palabras que hacen ping y baffle cuando son llamados.

- b. Dibujar un diagrama de la pila que muestre el estado del programa la primera vez que ping es llamado.

```
public static void zoop () {
    baffle ();
    System.out.print ("Vos zacata ");
    baffle ();
}

public static void main (String[] args) {
    System.out.print ("No, yo ");
    zoop ();
    System.out.print ("Yo ");
    baffle ();
}

public static void baffle () {
    System.out.print ("pac");
    ping ();
}
```



```
public static void ping () {
    System.out.println (".");
}
```

Ejercicio 3.3

El objetivo de este ejercicio es el de asegurarte que entiendas cómo escribir y llamar métodos que toman parámetros.

- Escribir la primera línea de un método llamado `zool` que toma tres parámetros: un `int` y dos `Strings`.
- Escribir una línea de código que invoque a `zool`, pasando como argumentos al valor 11, el nombre de tu primera mascota y el nombre de la calle en la cual creciste.

Ejercicio 3.4

El propósito de este ejercicio es que tomes código de un ejercicio anterior y lo encapsules en un método que toma parámetros. Deberías comenzar con una solución al Ejercicio 2.1.

- Escribir un método llamado `imprimirNorteamericano` que toma el día de la semana, día del mes, mes y año como parámetros y que imprime en el formato norteamericano.
- Probar el método invocándolo desde `main` y pasando argumentos apropiados. La salida debería verse parecido a esto (excepto que la fecha puede ser diferente):

Miércoles, Septiembre 29, 2010

- Una vez que hayas depurado `imprimirNorteamericano`, escribí otro método llamado `imprimirSudamericano` que imprima la fecha en el formato Sudamericano.

Ejercicio 3.5

Muchos cálculos pueden ser expresados de manera concisa usando la operación “multsuma”, que toma tres operandos y computa $a * b + c$. Algunos procesadores incluso proveen una implementación de hardware para esta operación para números de punto flotante.

- Crear un nuevo programa llamado `Multsuma.java`.
- Escribir un método llamado `multsuma` que toma tres `doubles` como parámetros y que imprime el resultado de multisumarlo.

- c. Escribir un método `main` que teste `multsuma` invocándolo con unos pocos parámetros simples, como por ejemplo 1.0, 2.0, 3.0, y después imprimir el resultado, que en ese caso debería ser 5.0.
- d. Además, en `main`, usar `multsuma` para computar los siguientes valores:

$$\sin \frac{\pi}{4} + \frac{\cos \frac{\pi}{4}}{2}$$

$$\log 10 + \log 20$$

- e. Escribir un método llamado `caramba` que toma un `double` como parámetro y que usa `multsuma` para calcular e imprimir

$$xe^{-x} + \sqrt{1 - e^{-x}}$$

AYUDA: el método de `Math` para elevar e a una potencia es `Math.exp`.

En la última parte, tenés la posibilidad de escribir un método que invoque a otro método que hayas escrito. Cada vez que hagas eso, es una buena idea probar el primer método cuidadosamente antes de que empieces a trabajar en el segundo. De otra manera, podrías encontrarte depurando dos métodos al mismo tiempo, lo cual puede ser muy difícil.

Uno de los propósitos de este ejercicio es que practiques reconocimiento de patrones: la habilidad de reconocer un problema específico como una instancia de una categoría general de problemas.

Capítulo 4

Condicionales y recursión

4.1 El operador módulo

El operador módulo trabaja con enteros (y expresiones enteras) y devuelve el *resto* de dividir el primer operando por el segundo. En Java, el operador módulo es un signo por ciento, %. La sintaxis es exactamente la misma que para los demás operadores:

```
int cociente = 7 / 3;  
int resto = 7 % 3;
```

El primer operador, división entera, devuelve 2. El segundo operador devuelve 1. Así, 7 dividido 3 es 2 con 1 de resto.

El operador módulo resulta ser sorprendentemente útil. Por ejemplo, podés chequear si un número es divisible por otro: si $x \% y$ es cero, entonces x es divisible por y .

Además, podés usar el operador módulo para extraer el dígito o los dígitos de más a la derecha. Por ejemplo, $x \% 10$ da el dígito más a la derecha de x (en base 10). De manera similar, $x \% 100$ da los últimos dos dígitos.

4.2 Ejecución condicional

A fin de escribir programas útiles, casi siempre necesitamos poder chequear ciertas condiciones, y cambiar el comportamiento del programa adecuadamente. Las **sentencias condicionales** nos permiten hacer justamente eso. La forma más simple es la sentencia `if`:

```
if (x > 0) {  
    System.out.println ("x es positivo");  
}
```

La expresión entre paréntesis se llama condición. Si es verdadera, entonces se ejecutan las sentencias entre llaves. Si la condición no es verdadera, nada sucede.

La condición puede contener cualquiera de los operadores de comparación, a veces llamados **operadores relacionales**:

<code>x == y</code>	<code>// x igual y</code>
<code>x != y</code>	<code>// x es distinto de y</code>
<code>x > y</code>	<code>// x es mayor que y</code>
<code>x < y</code>	<code>// x es menor que y</code>
<code>x >= y</code>	<code>// x es mayor o igual que y</code>
<code>x <= y</code>	<code>// x es menor o igual que y</code>

A pesar de que estas operaciones te son probablemente familiares, la sintaxis que usa Java es un poco diferente de los símbolos matemáticos como $=$, \neq y \leq . Un error común es usar un `=` simple en lugar del doble (`==`). No olvides que el `=` es el operador de asignación, y el `==` es el operador de comparación. Además, no existen cosas como `= < o >`.

Ambos lados de un operador condicional tienen que ser del mismo tipo. Sólo se pueden comparar ints con ints y doubles con doubles. Desafortunadamente, a esta altura no podrás comparar Strings de ninguna manera. Hay una forma de comparar Strings, pero no vamos a llegar a eso por un par de capítulos.

4.3 Ejecución alternativa

Una segunda forma de ejecución condicional es la ejecución alternativa, en la que hay dos posibilidades, y la condición determina cuál se ejecuta. La sintaxis se ve como:

```
if (x%2 == 0) {
    System.out.println ("x es par");
} else {
    System.out.println ("x es impar");
}
```

Si el resto de dividir `x` por 2 es cero, entonces sabemos que `x` es par, y este código imprime un mensaje indicando eso. Si la condición es falsa, se ejecuta la segunda sentencia de impresión. Dado que la condición tiene que ser verdadera o falsa, exactamente una de las alternativas será ejecutada.

Por otro lado, si pensás que vas a querer chequear la paridad (paridad o imparidad) de los números frecuentemente, tal vez prefieras “encapsular” este código en un método, del siguiente modo:

```

public static void imprimirParidad (int x) {
    if (x%2 == 0) {
        System.out.println ("x es par");
    } else {
        System.out.println ("x es impar");
    }
}

```

Ahora tenés un método llamado `imprimirParidad` que imprimirá un mensaje apropiado para cualquier entero que le pases. En el `main` llamarías a este método así:

```
imprimirParidad (17);
```

Recordá siempre que cuando *llamás* a un método, no tenés que declarar los tipos de los parámetros que le pases. Java puede darse cuenta de cuáles son sus tipos. Debés resistir la tentación de escribir cosas como:

```

int numero = 17;
imprimirParidad (int numero);           // MAL!!!

```

4.4 Condicionales encadenados

A veces te interesa chequear una cierta cantidad de condiciones relacionadas y elegir una entre varias acciones. Una manera de hacer esto es **encadenando** una serie de `ifs` y `elses`:

```

if (x > 0) {
    System.out.println ("x es positivo");
} else if (x < 0) {
    System.out.println ("x es negativo");
} else {
    System.out.println ("x es cero");
}

```

Estos encadenamientos pueden ser tan largos como quieras, aunque pueden ser difíciles de leer si se te van de las manos. Una manera de hacerlos más fáciles de leer es usando tabulación estándar, como se muestra en este ejemplo. Si mantenés todas las sentencias y las llaves alineadas, sos menos propenso a cometer errores de sintaxis y si los cometés, los vas a poder encontrar más rápidamente.

4.5 Condicionales anidados

Además del encadenamiento, es también posible anidar un condicional dentro de otro. Podríamos haber escrito el ejemplo anterior así:

```
if (x == 0) {
    System.out.println ("x es cero");
} else {
    if (x > 0) {
        System.out.println ("x es positivo");
    } else {
        System.out.println ("x es negativo");
    }
}
```

Ahora hay un condicional externo que contiene dos ramas. La primera rama contiene simplemente una sentencia print, pero la segunda rama contiene otra condicional, que tiene a su vez, dos ramas propias. Afortunadamente, esas dos ramas son ambas sentencias print, sin embargo, podrían haber sido sentencias condicionales también.

Nótese de nuevo que la tabulación ayuda a visualizar la estructura, pero de todos modos, los condicionales anidados se tornan difíciles de leer muy rápidamente. En general, es una buena idea evitarlos cuando se pueda.

Por otra parte, esta especie de **estructura anidada** es común, y la veremos de nuevo, así que será mejor que te vayas acostumbrando a ella.

4.6 La sentencia return

La sentencia return permite terminar la ejecución de un método antes de alcanzar el final. Una razón para usarlo es si detectas una condición de error:

```
public static void imprimirLogaritmo(double x) {
    if (x <= 0.0) {
        System.out.println ("Solo numeros positivos, por favor.");
        return;
    }

    double resultado = Math.log (x);
    System.out.println ("El logaritmo de x es " + resultado);
}
```

Esto define un método llamado imprimirLogaritmo que toma un double llamado x como parámetro. Lo primero que hace es chequear si x es menor o igual que cero, en cuyo caso imprime un mensaje de error y luego

usa la sentencia `return` para salir del método. El flujo de ejecución inmediatamente retorna al llamador y las líneas restantes del método no son ejecutadas.

Uso un valor de punto-flotante en el lado derecho de la condición porque hay una variable de punto-flotante en el izquierdo.

4.7 Conversión de tipos

Te podrás preguntar cómo puede ser que funcione una expresión como “El logaritmo de `x` es ” + `result`, dado que uno de los operandos es un `String` y el otro es un `double`. Bueno, en este caso Java está siendo inteligente por vos, convirtiendo automáticamente el `double` a un `String` antes de llevar a cabo la concatenación.

Esta especie de características es un ejemplo de un problema común en el diseño de lenguajes de programación, que consiste en que hay un conflicto entre *formalismo*, que es el requerimiento que dice que los lenguajes formales deberían tener reglas simples con pocas excepciones, y *conveniencia*, que es el requerimiento de que los lenguajes de programación sean sencillos de usar en la práctica.

En la mayoría de los casos, gana la conveniencia, lo que es usualmente bueno para los programadores experimentados (quienes se libran del formalismo riguroso pero difícil de manejar), pero malo para los programadores principiantes, que a menudo están abrumados por la complejidad de las reglas y la gran cantidad de excepciones. En este libro he intentado simplificar las cosas haciendo énfasis en las reglas y omitiendo muchas de las excepciones.

De todos modos, es práctico saber que cada vez que intentes “sumar” dos expresiones, si una de ellas es una `String`, entonces Java convertirá la otra en `String` y luego realizará una concatenación de cadenas. ¿Qué creés que ocurre si realizás una operación entre un entero y un valor de punto-flotante?

4.8 Recursión

He mencionado en el capítulo anterior que es posible que un método llame a otro, y hemos visto varios ejemplos de eso. Omití mencionar que también es posible que un método se llame a sí mismo. Puede no ser obvio por qué eso es algo bueno, pero resulta ser una de las cosas más mágicas e interesantes que un programa puede hacer. Por ejemplo, veamos el siguiente método:


```

public static void cuentaRegresiva (int n) {
    if (n == 0) {
        System.out.println ("Explosión!");
    } else {
        System.out.println (n);
        cuentaRegresiva (n-1);
    }
}

```

El nombre del método es `cuentaRegresiva` y toma un solo entero como parámetro. Si el parámetro es cero, imprime la palabra “Explosión.” En caso contrario, imprime el número y llama al método `cuentaRegresiva`—él mismo—pasando `n-1` como parámetro.

Te imaginás qué sucede si llamamos a este método en el `main`, así:

```
cuentaRegresiva (3);
```

La ejecución de `cuentaRegresiva` comienza con `n=3`, y dado que `n` no es cero, imprime el valor 3, y luego se llama a sí mismo...

La ejecución de `cuentaRegresiva` comienza con `n=2`, y dado que `n` no es cero, imprime el valor 2, y luego se llama a sí mismo...

La ejecución de `cuentaRegresiva` comienza con `n=1`, y dado que `n` no es cero, imprime el valor 1, y luego se llama a sí mismo...

La ejecución de `cuentaRegresiva` comienza con `n=0`, y dado que `n` es cero, imprime la palabra “Explosión!” y luego retorna al llamador.

La cuenta regresiva que tuvo `n=1` retorna al llamador.

La cuenta regresiva que tuvo `n=2` retorna al llamador.

La cuenta regresiva que tuvo `n=3` retorna al llamador.

Finalmente estamos de vuelta en `main` (qué viajecito). Entonces la salida total se ve como:

```

3
2
1
Explosion!

```

Como otro ejemplo, revisemos los métodos `nuevaLinea` y `tresLineas`.

```
public static void nuevaLinea () {  
    System.out.println ("");  
}  
  
public static void tresLineas () {  
    nuevaLinea (); nuevaLinea (); nuevaLinea ();  
}
```

Aunque estas sirven, no serían de mucha utilidad si yo quisiera imprimir 2 nuevas líneas, o 106. Una mejor alternativa sería

```
public static void nLineas (int n) {  
    if (n > 0) {  
        System.out.println ("");  
        nLineas (n-1);  
    }  
}
```

Este programa es muy similar; siempre y cuando `n` es mayor que cero, imprime una nueva línea, y luego se llama a sí mismo para imprimir `n-1` nuevas líneas adicionales. Por lo tanto, el número total de nuevas líneas que se imprimen es $1 + (n-1)$, que usualmente termina siendo `n`.

El proceso de que un método se llame a sí mismo se llama **recursión**, y a dichos métodos se les dice ser **recursivos**.

4.9 Diagramas de la pila de ejecución 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 diagramas pueden facilitar la interpretación de un método recursivo.

Recordá que cada vez que un método es llamado, crea una nueva instancia del método, la cual contiene una nueva versión de las variables locales y los parámetros del programa.

La siguiente figura es un diagrama de pila de cuentaRegresiva, llamado con `n = 3`:



Hay una instancia del `main` y cuatro instancias de `cuentaRegresiva`, cada una con un valor diferente del parámetro `n`. Al final de la pila, está el llamado a `cuentaRegresiva` con `n=0`, el cual es el caso base. Éste no hace un llamado recursivo, así que no hay más instancias de `cuentaRegresiva`.

La instancia del `main` está vacía porque `main` no tiene ningún parámetro ni variables locales.

Ejercicio 4.1

Dibujá un diagrama de pila que muestre el estado del programa después de que `main` llamó a `nLines` con parámetro `n=4`, justo antes de la última instancia de `nLines` retorne.

4.10 Convención y Ley Divina

En las últimas secciones, utilicé la frase “por convención” varias veces para indicar decisiones de diseño que son arbitrarias en el sentido de que no hay razones significativas para hacer las cosas de una manera en lugar de otra, pero sí dictadas por convención.

En estos casos, te va a convenir estar familiarizado con las convenciones y usarlas, dado que hará que tu programa sea más fácil de entender por otros. Al mismo tiempo, es importante distinguir entre (al menos) tres clases de reglas:

Ley Divina: Esta es mi frase para referirme a una regla que es cierta por algún principio subyacente de la lógica o la matemática, y que es cierta en cualquier lenguaje de programación (o algún otro sistema formal). Por ejemplo, no hay manera de especificar un cuadro delimitador usando menos de cuatro piezas de información. Otro ejemplo es que la suma de enteros es conmutativa. Eso es parte de la definición de suma y nada tiene que ver con Java.

Reglas de Java: Estas son reglas sintácticas y semánticas de Java que no podés violar, porque el programa resultante no compilará o no correrá. Algunas son arbitrarias; por ejemplo, el hecho de que el símbolo + representa suma y concatenación de cadenas. Otras reflejan limitaciones subyacentes del proceso de compilación o de ejecución. Por ejemplo, tenés que especificar los tipos de los parámetros al definir un método pero no debés hacerlo al llamarlo.

Estilo y convención: Hay muchas reglas que no son requeridas por el compilador, pero que son esenciales para escribir programas correctos, que se puedan depurar y modificar, y que otras personas puedan leer. Es el caso de la tabulación y ubicación de las llaves, como también las convenciones para nombrar variables, métodos y clases.

A medida que avancemos, intentaré indicar en qué categoría cae cada cosa, pero tal vez quieras pensarlo vos mismo cada tanto.

Como estamos en tema, habrás podido notar que los nombres de las clases siempre comienzan en mayúscula, pero las variables y los métodos comienzan en minúscula. Si un nombre tiene más de una palabra, usualmente se escribe en mayúscula sólo la primera letra de cada palabra, como en `nuevaLinea` e `imprimirParidad`. ¿En qué categoría cae esta regla?

4.11 Glosario

módulo: Operador que trabaja con enteros y devuelve el resto de una división. En Java se denota con un signo porcentual (%).

condicional: Bloque de sentencias que podría ejecutarse (o no) dependiendo de alguna condición.

encadenamiento: Manera de unir varias sentencias condicionales en secuencia.

anidamiento: El hecho de poner una sentencia condicional adentro de una o ambas ramas de otra sentencia condicional.

coordenada: Variable o valor que especifica una ubicación en dos dimensiones en una ventana gráfica.

pixel: Unidad en la que se miden las coordenadas.

cuadro delimitador Manera común de especificar las coordenadas de un área rectangular.

casteo de tipos: Operación que convierte un tipo en otro. En Java aparece como un nombre de tipo entre paréntesis, como (int).

interfaz: Descripción de los parámetros requeridos por un método y sus tipos.

prototipo: Una manera de describir la interfaz de un método usando sintaxis estilo Java.

recursión: Proceso de llamar al mismo método que se está corriendo actualmente.

recursión infinita: Método que se llama a sí mismo recursivamente sin llegar a su caso base nunca. El caso usual resulta en una excepción de tipo StackOverflowException (Pila).

fractal: Clase de imagen que se define recursivamente, de modo que cada parte de la imagen es una versión más pequeña del todo.

4.12 Ejercicios

Ejercicio 4.2

Este ejercicio revisa el flujo de ejecución a través de un programa con múltiples métodos. Lee el siguiente código y responde las preguntas de abajo.

```
public class Zumbido {

    public static void desconcertar (String dirigible) {
        System.out.println (dirigible);
        sipo ("ping", -5);
    }

    public static void sipo (String membrillo, int flag) {
        if (flag < 0) {
            System.out.println (membrillo + " sup");
        } else {
            System.out.println ("ik");
            desconcertar (membrillo);
            System.out.println ("muaa-ja-ja-ja");
        }
    }

    public static void main (String[] args) {
        sipo ("traqueteo", 13);
    }
}
```

- a. Escribí el número 1 al lado de la primera sentencia del programa que será ejecutada. Tené cuidado de distinguir aquellas cosas que sean sentencias de aquellas que no.
- b. Escribí el número 2 al lado de la segunda sentencia, y así siguiendo hasta el final del programa. Si una sentencia se ejecuta más de una vez, puede que termine con más de un número al lado.
- c. ¿Cuáles es el valor del parámetro dirigible cuando se llama a desconcertar?
- d. ¿Cuál es la salida de este programa?

Ejercicio 4.3

La primera estrofa de la canción “99 Botellas de Cerveza” dice:

Hay 99 botellas de cerveza en la pared, hay 99 botellas de cerveza,
una sola agarrás, y después la pasás, hay 98 botellas de cerveza en la
pared.

Las estrofas siguientes son idénticas excepto por el número de botellas que va haciéndose menor en uno en cada estrofa, hasta que el último verso dice:

No hay más botellas de cerveza en la pared, no hay más botellas de
cerveza, no las agarrarás, y no las pasarás, porque no hay más botel-
las de cerveza en la pared.

Y luego la canción (por fin) termina.

Escribí un programa que imprima la letra completa de “99 Botellas de Cerveza.” Tu programa debería incluir un método recursivo que realice la parte difícil, pero también puede que quieras escribir métodos adicionales para separar las diferentes funcionalidades del programa.

A medida que desarrolles tu código, probablemente quieras testearlo con un número más bajo de estrofas, como “3 Botellas de Cerveza.”

El propósito de este ejercicio es tomar un problema y partirlo en problemas más pequeños, y resolver los problemas pequeños escribiendo métodos simples y sencillos de depurar.

Ejercicio 4.4

¿Cuál es la salida del siguiente programa?

```
public class Narf {
    public static void sup (String pepe, int bruno) {
        System.out.println (pepe);
        if (bruno == 5)
            ping ("no ");
        else
            System.out.println ("!");
    }
}
```

```

public static void main (String[] args) {
    int bis = 5;
    int bas = 2;
    sup("solo por", bis);
    clink (2*bas);
}

public static void clink (int tenedor) {
    System.out.print ("Está el ");
    sup("desayuno ", tenedor) ;
}

public static void ping (String cuerda) {
    System.out.println ("cualquiera " + cuerda + "mas ");
}
}

```

Ejercicio 4.5

El Último Teorema de Fermat dice que no hay enteros a , b , y c tales que

$$a^n + b^n = c^n$$

excepto en el caso en que $n = 2$.

Escribí un método llamado `chequearFermat` que tome cuatro enteros como parámetros— a , b , c y n —y que chequee si el teorema de Fermat se cumple. Si n es mayor que 2 y sucede que es cierto que $a^n + b^n = c^n$, el programa debería imprimir “Recorcholis, Fermat estaba equivocado!” De lo contrario, el programa debería imprimir “No, eso no funciona.”

Podes asumir que hay un método llamado `elevaAPotencia` que toma dos enteros como parámetros y eleva el primero a la potencia que indica el segundo. Por ejemplo:

```
int x = elevaAPotencia (2, 3);
```

asignará el valor 8 a x , porque $2^3 = 8$.

Capítulo 5

Métodos con resultados

5.1 Valores de retorno

Algunos de los métodos preincorporados que estuvimos usando, como las funciones de `Math`, producen resultados. Es decir, el efecto de llamar al método es la generación de un nuevo valor, el cual usualmente asignamos a una variable o usamos como parte de una expresión. Por ejemplo:

```
double e = Math.exp (1.0);  
double altura = radio * Math.sin (angulo);
```

Pero hasta ahora todos los métodos que escribimos han sido métodos **void**; es decir, métodos que no devuelven ningún valor. Cuando llamamos un método `void`, lo hacemos en general en una línea aislada, sin asignaciones:

```
nLineas (3);  
g.drawOval (0, 0, ancho, alto);
```

En este capítulo, vamos a escribir métodos que devuelven cosas, a los cuales me referiré como métodos **con resultados**, a falta de un mejor nombre. El primer ejemplo es `area`, que toma un `double` como parámetro y devuelve el área de un círculo con el radio dado:

```
public static double area (double radio) {  
    double area = Math.PI * radio * radio;  
    return area;  
}
```


Lo primero que notaremos es que el comienzo de la definición del método es diferente. En lugar de `public static void`, lo que indica un método `void`, vemos `public static double`, lo que indica que el valor de retorno de este método tendrá tipo `double`. Todavía no te expliqué lo que significa `public static`, pero tené paciencia.

Notá también que la última línea es una forma alternativa de la sentencia `return` que incluye un valor de retorno. Esta sentencia significa, “Salí inmediatamente de este método y usá la siguiente expresión como valor de retorno”. La expresión que proveas puede ser arbitrariamente complicada, con lo que podríamos haber escrito este método más conciso:

```
public static double area (double radio) {  
    return Math.PI * radio * radio;  
}
```

Por otro lado, las variables **temporales** como `area` suelen simplificar la depuración. En cualquier caso, el tipo de la expresión junto a la sentencia `return` debe coincidir con el tipo de retorno del método. En otras palabras, al declarar que el valor de retorno es de tipo `double`, estás haciendo la promesa de que este método producirá un `double`. Si escribimos un `return` sin una expresión, o una expresión de un tipo incorrecto, el compilador nos dirá que lo corriamos.

A veces es útil tener múltiples sentencias `return`, una en cada rama de un condicional:

```
public static double valorAbsoluto (double x) {  
    if (x < 0) {  
        return -x;  
    } else {  
        return x;  
    }  
}
```

Como estas sentencias `return` están en un condicional, sólo una de ellas se ejecutará. Aunque es válido tener más de una sentencia `return` en un método, tenemos que tener en cuenta que tan pronto se ejecute una de ellas, el método termina sin ejecutar ninguna sentencia posterior.

El código que aparece a continuación de una sentencia `return`, o en cualquier otro lugar donde nunca pueda ser ejecutado, se llama **código muerto**. Algunos compiladores te advierten si una parte de tu código es código muerto.

Si ponemos sentencias `return` dentro de un condicional, entonces tenemos que garantizar que *todo camino posible* del método da con una sentencia `return`. Por ejemplo:

```

public static double valorAbsoluto (double x) {
    if (x < 0) {
        return -x;
    } else if (x > 0) {
        return x;
    }
    // INCORRECTO!!
}

```

Este programa no es válido porque si x llegara a ser 0, entonces ninguna de las condiciones sería verdadera y el método terminaría sin dar con ninguna sentencia return. Un típico mensaje del compilador en este caso sería “return statement required in absoluteValue”¹, el cual es un mensaje confuso considerando que ya hay dos sentencias return en el método.

5.2 Desarrollo de un programa

A esta altura deberías ser capaz de mirar cualquier método de Java y decir qué es lo que hace. Pero puede no estar claro aun cómo escribirlos. Te voy a sugerir una técnica a la que yo llamo **desarrollo incremental**.

Como ejemplo, imaginemos que queremos hallar la distancia entre dos puntos, dados por las coordenadas (x_1, y_1) y (x_2, y_2) . Siguiendo la definición usual,

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

El primer paso sería considerar cómo debería verse un método distancia en Java. Es decir, cuáles serían los valores de entrada (parámetros) y cuál el de salida (valor de retorno).

En este caso, los dos puntos son los parámetros, y es natural representarlos usando cuatro doubles, aunque veremos más adelante que existe un objeto `Point`² en Java que podríamos usar. El valor de retorno es la distancia, la cual tendrá tipo double.

Ya podemos escribir un esbozo del método:

```

public static double distancia (double x1, double y1,
                                double x2, double y2) {
    return 0.0;
}

```

La sentencia `return 0.0;` es un relleno necesario para que el programa compile. Obviamente, hasta aquí el programa no hace nada útil, pero vale la pena intentar compilarlo para poder identificar cualquier error de sintaxis antes de hacerlo más complicado.

1. N.d.T.: “sentencia return requerida en valorAbsoluto”.

2. N.d.T.: *point* significa *punto* en inglés.

Para probar el nuevo método tenemos que llamarlo con valores de prueba. En algún lugar dentro del main agregaríamos:

```
double dist = distancia (1.0, 2.0, 4.0, 6.0);
```

Elegí estos valores para que la distancia horizontal sea 3 y la distancia vertical sea 4; así, el resultado será 5 (la hipotenusa de un triángulo 3-4-5). Cuando estás probando un método es útil conocer la respuesta correcta.

Una vez que verificamos la sintaxis de la definición del método, podemos empezar a agregar líneas de código una a la vez. Luego de cada cambio incremental recompilamos y ejecutamos el programa. De esta manera, en cualquier momento sabemos exactamente dónde debe estar el error—en la última línea que agregamos.

El siguiente paso para el cálculo es hallar las diferencias $x_2 - x_1$ e $y_2 - y_1$. Guardaremos estos valores en variables temporales a las que llamaremos dx y dy.

```
public static double distancia (double x1, double y1,
                                double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    System.out.println ("dx es " + dx);
    System.out.println ("dy es " + dy);
    return 0.0;
}
```

Agregué sentencias de impresión que me dejarán verificar los valores intermedios. Como dijimos, ya sabemos que estos deben ser 3.0 y 4.0.

Cuando terminemos con el método, sacaremos las sentencias de impresión. A este tipo de código se lo llama **código de soporte**, porque ayuda a construir el programa pero no es parte del producto final. A veces es una buena idea conservar el código de soporte pero comentándolo, por si acaso lo necesitamos más tarde.

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

```
public static double distancia (double x1, double y1,
                                double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dcuadrado = dx*dx + dy*dy;
    System.out.println ("dcuadrado es " + dcuadrado);
    return 0.0;
}
```

Nuevamente, compilamos y ejecutamos el programa hasta aquí y verificamos el valor intermedio (el cual debería ser 25.0).

Finalmente, podemos usar el método `Math.sqrt` para calcular y devolver el resultado.

```
public static double distancia (double x1, double y1,  
                                double x2, double y2) {  
    double dx = x2 - x1;  
    double dy = y2 - y1;  
    double dcuadrado = dx*dx + dy*dy;  
    double resultado = Math.sqrt (dcuadrado);  
    return resultado;  
}
```

Luego, en el `main`, deberíamos imprimir y verificar el valor del resultado.

A medida que vayas ganando experiencia programando, vas a poder escribir y depurar más de una línea a la vez. De todas formas, este proceso de desarrollo incremental puede ahorrarte mucho tiempo de depuración.

Los aspectos clave del proceso son:

- Empezar con un programa que funcione e ir haciendo pequeños cambios incrementalmente. En cualquier momento, si hay un error, sabrás exactamente dónde está.
- Usar variables temporales para almacenar valores intermedios para poder así imprimirlos y verificarlos.
- Una vez que el programa esté funcionando, tal vez quieras eliminar el código de soporte o consolidar muchas sentencias usando expresiones compuestas, pero sólo si esto no hace que el programa sea difícil de leer.

5.3 Composición

Tal como uno esperaría, una vez que definís un nuevo método, podés usarlo como parte de una expresión, y podés construir nuevos métodos usando métodos existentes. Por ejemplo, ¿qué harías si alguien te diera dos puntos, el centro de un círculo y un punto del perímetro, y te pidiera el área del círculo?

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

```
double radio = distancia (xc, yc, xp, yp);
```

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

```
double area = area (radio);  
return area;
```

Encapsulando todo en un método, tenemos:

```
public static double pepe (double xc, double yc,  
                           double xp, double yp) {  
    double radio = distancia (xc, yc, xp, yp);  
    double area = area (radio);  
    return area;  
}
```

El nombre de este método es pepe, lo cual puede parecer algo raro. Explicaré el porqué de esto en la siguiente sección.

Las variables temporales radio y area son útiles para el desarrollo y la depuración, pero una vez que el programa está funcionando podemos hacerlo más conciso componiendo las llamadas a los métodos:

```
public static double pepe (double xc, double yc,  
                           double xp, double yp) {  
    return area (distancia (xc, yc, xp, yp));  
}
```

5.4 Sobrecarga

En la sección anterior probablemente notaste que pepe y area realizan funciones similares—hallar el área de un círculo—pero toman diferentes parámetros. Para área, debemos proveer el radio; para pepe proveemos dos puntos.

Si dos métodos hacen lo mismo, es natural llamarlos por el mismo nombre. Es decir, tendría más sentido si pepe se llamara area.

Tener más de un método con el mismo nombre, lo cual se conoce como **sobrecarga**, es válido en Java *siempre y cuando cada versión del método tome diferentes parámetros*. Con lo cual podemos permitirnos renombrar pepe:

```
public static double area (double x1, double y1,  
                           double x2, double y2) {  
    return area (distancia (xc, yc, xp, yp));  
}
```

Cuando se llama un método sobrecargado, Java sabe qué versión se quiere usar inspeccionando los argumentos que se le pasan. Si escribimos:

```
double x = area (3.0);
```

Java busca un método llamado `area` que recibe un solo `double` como argumento, y por lo tanto usa la primera versión, que interpreta el argumento como el radio. Si escribimos:

```
double x = area (1.0, 2.0, 4.0, 6.0);
```

Java usa la segunda versión de `area`. Más interesante aun, la segunda versión de `area` llama efectivamente a la primera.

Muchos de los comandos preincorporados de Java están sobrecargados, indicando que existen diferentes versiones que aceptan distinto número o tipos de parámetros. Por ejemplo, hay versiones de `print` y de `println` que aceptan un único parámetro de cualquier tipo. En la clase `Math`, hay una versión de `abs` que funciona con `doubles` y otra versión para `ints`.

Si bien la sobrecarga es una herramienta muy útil, debe ser usada con cuidado. Podés llegar a confundirte mucho si intentás depurar una versión de un método llamando accidentalmente a otra.

En efecto, eso me recuerda una de las reglas principales de la depuración: **¡asegurate que la versión del programa que estás mirando es la versión del programa que se está ejecutando!** A veces, podés encontrarte haciendo un cambio tras otro en tu programa y viendo el mismo resultado en cada ejecución. Esta es una advertencia de que por uno u otro motivo no estás ejecutando la versión del programa que crees estar ejecutando. Para verificarlo, insertá una sentencia `print` (no importa lo que imprimas) y asegurate que el comportamiento del programa cambia acorde a eso.

5.5 Expresiones booleanas

Muchas de las operaciones que hemos visto producen resultados que son del mismo tipo que el de sus operandos. Por ejemplo, el operador `+` toma dos `ints` y produce un `int`, o dos `doubles` y produce un `double`, etc.

Las excepciones que hemos visto son los **operadores relacionales**, que comparan `ints` y `floats` y devuelven `true` o `false`. `true` y `false` son valores especiales en Java, y juntos conforman un tipo llamado **boolean**. Seguramente recordás que cuando definimos los tipos, dijimos que eran conjuntos de valores. En el caso de los `ints`, `doubles` y `Strings`, los conjuntos son bastante grandes. Para los `booleans`, no demasiado.

Las variables y expresiones booleanas funcionan exactamente igual que otro tipo de variables y expresiones:

```
boolean pepe;  
pepe = true;  
boolean resultado = false;
```

El primer ejemplo es una simple declaración de una variable; el segundo ejemplo es una asignación y el tercero es una combinación de una declaración y una asignación, llamada a veces **inicialización**. Los valores `true` y `false` son palabras reservadas en Java así que pueden aparecer con un color diferente, dependiendo de tu entorno de desarrollo.

Tal como mencioné, el resultado de un operador relacional es un booleano, así que podés almacenar el resultado de una comparación en una variable:

```
boolean esPar = (n%2 == 0); // verdadero si n es par  
boolean esPositivo = (x > 0); // verdadero si x es positivo
```

y luego usarlo como parte de una sentencia condicional más adelante:

```
if (esPar) {  
    System.out.println ("n era par cuando lo revisé");  
}
```

Una variable usada de esta forma es usualmente llamada una **bandera**, ya que marca la presencia o ausencia de cierta condición.

5.6 Operadores lógicos

En Java existen tres **operadores lógicos**: Y, O y NO, los cuales se notan usando los símbolos `&&`, `||` y `!`, respectivamente. La semántica (significado) de estos operadores es similar a su significado en castellano. Por ejemplo `x > 0 && x < 10` es verdadero sólo cuando `x` es mayor que cero Y menor que 10.

La expresión `esPar || n%3 == 0` es verdadera si *alguna* de las dos condiciones es verdadera, es decir, si `esPar` es verdadera O si el número `n` es divisible por 3.

Finalmente, el operador NO tiene el efecto de negar o invertir una expresión booleana, con lo cual `!esPar` es verdadero si `esPar` es `false`—si el número es impar.

Los operadores lógicos proveen formas de simplificar las sentencias condicionales anidadas. Por ejemplo, ¿cómo escribirías el siguiente código usando un sólo condicional?

```

if (x > 0) {
    if (x < 10) {
        System.out.println ("x es un dígito simple positivo.");
    }
}

```

5.7 Métodos booleanos

Los métodos pueden devolver valores booleanos al igual que de cualquier otro tipo, lo cual suele ser conveniente para ocultar comprobaciones complejas dentro de métodos. Por ejemplo:

```

public static boolean esDigitoSimple (int x) {
    if (x >= 0 && x < 10) {
        return true;
    } else {
        return false;
    }
}

```

El nombre de este método es `esDigitoSimple`. Es común dar a los métodos booleanos nombres que suenen como preguntas de tipo sí/no. El tipo de retorno es `boolean`, lo que indica que toda sentencia `return` debe proveer una expresión booleana.

El código en sí mismo es directo, aunque un poco más largo de lo necesario. Recordá que la expresión `x >= 0 && x < 10` tiene tipo booleano, entonces no hay nada de malo en devolverla directamente evitando a la vez la sentencia `if`:

```

public static boolean esDigitoSimple (int x) {
    return (x >= 0 && x < 10);
}

```

En el `main` podés llamar a este método de las formas usuales:

```

boolean esGrande = !esDigitoSimple (17);
System.out.println (esDigitoSimple (2));

```

La primera línea asigna el valor `true` a `esGrande` sólo si 17 *no es* un número de un solo dígito. La segunda línea imprime `true` ya que 2 es un número de un solo dígito. Así es, `println` está sobrecargada para manejar booleanos también.

El uso más común para los métodos booleanos es dentro de sentencias condicionales


```

if (esDigitoSimple (x)) {
    System.out.println ("x es pequeño");
} else {
    System.out.println ("x es grande");
}

```

5.8 Más recursión

Ahora que tenemos métodos que devuelven valores, te interesará saber que lo que tenemos es un lenguaje de programación **completo**, y con eso me refiero a que cualquier cosa que puede ser computada, puede ser expresada en este lenguaje. Cualquier programa alguna vez escrito se puede reescribir usando sólo las herramientas que hemos usado hasta ahora (en realidad, necesitaríamos algunos comandos para controlar dispositivos tales como el teclado, mouse, discos rígidos, etc., pero con eso basta).

Demostrar esta afirmación es algo para nada trivial y fue logrado por primera vez por Alan Turing, uno de los primeros científicos de la computación (bueno, muchos podrían decir que fue un matemático, pero muchos de los primeros científicos de la computación comenzaron como matemáticos). Apropiadamente, esto es conocido como la tesis de Turing. Si alguna vez tomás un curso de Teoría de la Computación vas a tener la oportunidad de ver la demostración.

Para darte una idea de lo que podés hacer con las herramientas aprendidas hasta ahora, veamos algunos métodos de evaluar funciones matemáticas definidas recursivamente. Una definición recursiva es similar a una definición circular, en el sentido en que la definición contiene una referencia a lo que se está definiendo. Una estricta definición circular es usualmente bastante inútil:

borroso: describe algo que es borroso.

Si vieras esa definición en un diccionario te irritarías. Por otro lado, si buscaras la definición de la función matemática **factorial**, verías algo como:

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

(El factorial usualmente se nota con el símbolo $!$, el cual no debe ser confundido con el operador lógico de Java $!$ que significa NO.) 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 $2!$, el cual es 2

por 1!, el cual es 1 por 0!. Juntando todo tenemos que 3! es igual a 3 por 2 por 1 por 1, lo que da 6.

Si podemos escribir una definición recursiva de algo, usualmente podemos escribir un programa de Java para evaluarlo. El primer paso es decidir cuáles son los parámetros para esta función, y de qué tipo es el valor de retorno. Pensando un poco, deberías concluir qué factorial toma un entero como parámetro y devuelve un entero:

```
public static int factorial (int n) {  
}
```

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

```
public static int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    }  
}
```

Si no, y esta es la parte interesante, tenemos que hacer un llamado recursivo para hallar el factorial de $n - 1$, y luego multiplicarlo por n .

```
public static int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        int recursion = factorial (n-1);  
        int resultado = n * recursion;  
        return resultado;  
    }  
}
```

Si miramos el flujo de la ejecución de este programa, es similar a n líneas del capítulo anterior. Si llamamos a factorial con el valor 3:

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

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

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

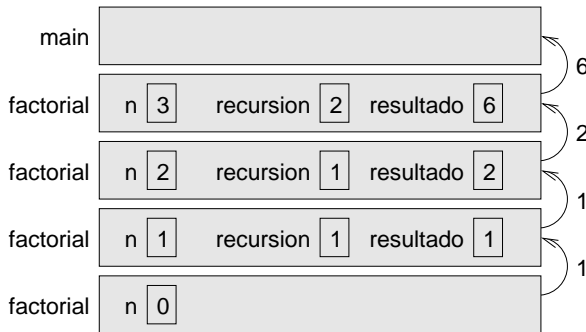
Como 0 es igual a cero, tomamos la primera rama del condicional y devolvemos el valor 1 inmediatamente, sin hacer más llamados recursivos.

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

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

El valor de retorno (2) se multiplica por n, que es 3, y se devuelve el resultado (6) al main, o a quien sea que haya llamado a factorial (3).

Así es como se vería el diagrama de la pila para esta secuencia de llamados a función:



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

Notemos que en la última instancia de factorial, las variables locales recursion y resultado no existen ya que cuando n=0 la rama que las genera no se ejecuta.

5.9 Salto de fe

Seguir el flujo de ejecución es una manera de leer programas, pero como viste en la sección anterior, puede volverse rápidamente laberíntico. Una alternativa es lo que yo llamo el “salto de fe”. Cuando llegamos a una llamada a un método, en lugar de seguir el flujo de ejecución, asumimos que el método funciona correctamente y devuelve el valor apropiado.

De hecho, ya estamos realizando este salto de fe cuando usamos los métodos preincorporados. Cuando llamamos a `Math.cos` o `drawOval`, no

examinamos las implementaciones de estos métodos. Simplemente asumimos que funcionan porque la gente que escribió las clases preincorporadas eran buenos programadores.

Bien, lo mismo sucede cuando invocás uno de tus métodos propios. Por ejemplo, en la Sección 5.7 escribimos el método `esDigitoSimple` que determina si un número está entre 0 y 9. Una vez que nos convencemos de que el método es correcto—mediante pruebas y examinación del código—podemos usarlo sin tener que mirar nunca más el código.

Lo mismo sucede con los programas recursivos. Cuando llegamos al llamado recursivo, en lugar de seguir el flujo de ejecución, *asumimos* que el llamado recursivo funciona (que llega al resultado correcto), y luego nos preguntamos, “Asumiendo que puedo hallar el factorial de $n-1$, ¿puedo calcular el factorial de n ?”. En este caso claramente se puede, multiplicándolo por n .

Está claro que es un poco raro asumir que el método funciona correctamente cuando aun ni siquiera lo terminamos de escribir, ¡pero por eso es que lo llamamos un salto de fe!

5.10 Un ejemplo más

En el ejemplo anterior usé variables temporales para detallar los pasos y para hacer el código más fácil de depurar, pero podría haberme ahorrado unas líneas:

```
public static int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial (n-1);  
    }  
}
```

De ahora en más, voy a usar versiones más concisas, pero te recomiendo usar versiones más explícitas mientras desarrolles código. Cuando lo tengas funcionando podés ajustarlo un poco, si te sentís inspirado.

Luego de `factorial`, el clásico ejemplo de una función matemática definida recursivamente es `fibonacci`, la cual se define de la siguiente manera:

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

Traducida a Java, esto es

```
public static int fibonacci (int n) {  
    if (n == 0 || n == 1) {  
        return 1;  
    } else {  
        return fibonacci (n-1) + fibonacci (n-2);  
    }  
}
```

Si intentás seguir el flujo de ejecución aquí, incluso para valores realmente pequeños de n , te explotará la cabeza. Pero siguiendo el salto de fe, si asumimos que los dos llamados recursivos (así es, se puede hacer dos llamados recursivos) funcionan correctamente, entonces queda claro que obtenemos el resultado correcto sumándolos.

5.11 Glosario

tipo de retorno: Parte en la declaración de un método que indica qué tipo de valor devuelve el mismo.

valor de retorno: Valor devuelto como resultado de una llamada a un método.

código muerto: Parte de un programa que no puede nunca ser ejecutada, usualmente porque aparece a continuación de una sentencia `return`.

código de soporte: Código usado durante el desarrollo de un programa pero que no es parte de la versión final.

void: Tipo de retorno especial que identifica a un método `void`; es decir, uno que no devuelve ningún valor.

sobrecarga: Tener más de un método con el mismo nombre pero con distintos parámetros. Cuando se llama a un método sobrecargado Java sabe qué versión usar examinando los argumentos provistos al mismo.

boolean: Tipo de variable que puede contener únicamente los dos valores `true` y `false`.

bandera: Variable (usualmente `boolean`) que registra una condición o información de estado.

operador condicional: Operador que compara dos valores y produce un booleano que indica la relación entre los operandos.

operador lógico: Operador que combina valores booleanos y produce resultados booleanos.

inicialización: Sentencia que declara una nueva variable y asigna al mismo tiempo un valor a ella.

5.12 Ejercicios

Ejercicio 5.1

Dados tres palitos, será posible o no disponerlos de tal manera que formen un triángulo. Por ejemplo, si uno de los palitos mide 12 centímetros de largo y los otros dos son de un centímetro, está claro que no es posible hacer que los dos palitos cortos se toquen en el centro. Para cualesquiera tres longitudes, existe un simple test para ver si es posible formar un triángulo:

“Si alguna de las tres longitudes es mayor que la suma de las otras dos, entonces no es posible formar un triángulo. En otro caso, siempre se puede.”

Escribí un método llamado `esTriangulo` que tome tres enteros como parámetros, y que devuelva `true` o `false`, dependiendo de si es posible formar un triángulo con las longitudes dadas.

El objetivo de este ejercicio es usar sentencias condicionales para escribir métodos que devuelven un valor.

Ejercicio 5.2

Escribir un método de clase llamado `esDivisible` que toma dos enteros, `n` y `m`, y devuelve verdadero si `n` es divisible por `m` y falso en caso contrario.

Ejercicio 5.3

¿Cuál es la salida del siguiente programa? El objetivo de este ejercicio es asegurar que entiendas los operadores lógicos y el flujo de ejecución en los métodos con resultados.

```
public static void main (String[] args) {  
    boolean bandera1 = esRaro (202);  
    boolean bandera2 = esBorroso (202);  
    System.out.println (bandera1);  
    System.out.println (bandera2);  
}
```

```

        if (bandera1 && bandera2) {
            System.out.println ("ping!");
        }
        if (bandera1 || bandera2) {
            System.out.println ("pong!");
        }
    }

    public static boolean esRaro (int x) {
        boolean banderaRaro;
        if (x%2 == 0) {
            banderaRaro = true;
        } else {
            banderaRaro = false;
        }
        return banderaRaro;
    }

    public static boolean esBorroso (int x) {
        boolean banderaBorroso;
        if (x > 0) {
            banderaBorroso = true;
        } else {
            banderaBorroso = false;
        }
        return banderaBorroso;
    }
}

```

Ejercicio 5.4

La distancia entre dos puntos (x_1, y_1) y (x_2, y_2) es

$$Distancia = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Escribir un método llamado `distancia` que tome cuatro doubles (x_1 , y_1 , x_2 e y_2) y que imprima la distancia entre los puntos.

Tenés que asumir que existe un método llamado `sumCuadrados` que calcula y devuelve la suma de los cuadrados de sus argumentos. Por ejemplo:

```
double x = sumCuadrados (3.0, 4.0);
```

asignará el valor 25.0 a `x`.

El objetivo de este ejercicio es escribir un método nuevo que use un método existente. Tenés que escribir sólo un método: `distancia`. No tenés que escribir `sumCuadrados` o `main` y no tenés que llamar a `distancia`.

Ejercicio 5.5

El objetivo de este ejercicio es practicar la sintaxis de los métodos con resultados.

- Recuperá tu solución para el Ejercicio 3.5 y asegurate de que aún compila y ejecuta.
- Transformá `multsuma` en un método con resultado, para que en lugar de imprimir un resultado, lo devuelva.
- En todos los lugares en el programa en donde se llame a `multsuma`, cambiá la llamada de manera tal que se almacene el resultado en una variable y/o se imprima el resultado.
- Transformá `caramba` de la misma manera.

Ejercicio 5.6

El objetivo de este ejercicio es usar un diagrama de pila para entender la ejecución de un programa recursivo.

```
public class Prod {  
  
    public static void main (String[] args) {  
        System.out.println (prod (1, 4));  
    }  
  
    public static int prod (int m, int n) {  
        if (m == n) {  
            return n;  
        } else {  
            int recursion = prod (m, n-1);  
            int resultado = n * recursion;  
            return resultado;  
        }  
    }  
}
```

- Dibujá un diagrama de pila mostrando el estado del programa justo antes de que la última instancia de `prod` se complete. ¿Cuál es la salida del programa?
- Explicá en pocas palabras qué hace `prod`.
- Reescribir `prod` sin usar las variables temporales `recursion` y `resultado`.

Ejercicio 5.7

El objetivo de este ejercicio es traducir una definición recursiva a un método de Java. La función Ackerman se define, para números no negativos, como sigue:

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0, n = 0 \\ A(m - 1, A(m, n - 1)) & \text{si } m > 0, n > 0 \end{cases}$$

Escribí un método llamado `ack` que tome dos ints como parámetros y que calcule y devuelva el valor de la función Ackerman para dichos valores.

Testeá tu implementación de Ackerman llamándola desde el `main` e imprimiendo los valores devueltos.

ADVERTENCIA: el valor de retorno se hace muy grande rápidamente. Deberías probarlo sólo para valores pequeños de m y n (no mayores que 2).

Ejercicio 5.8

- a. Creá un programa llamado `Recursion.java` y escribí en él los siguientes métodos:

```
// primero: devuelve el primer carácter de la cadena dada
public static char primero (String s) {
    return s.charAt (0);
}

// último: devuelve una nueva cadena que contiene todas
// las letras de la cadena dada, salvo la primera
public static String resto (String s) {
    return s.substring (1, s.length());
}

// largo: devuelve el largo de la cadena dada
public static int largo (String s) {
    return s.length();
}
```

- b. Escribí código en el `main` que testee cada uno de estos métodos. Asegurate que funcionan y que entendés qué hace cada uno de ellos.
- c. Escribí un método llamado `imprimirCadena` que tome una cadena como parámetro y que imprima las letras de la cadena, una en cada línea. Debe ser un método `void`.
- d. Escribí un método llamado `imprimirInverso` que haga lo mismo que el método `imprimirCadena` pero imprimiendo la cadena al revés (y un carácter por línea).
- e. Escribir un método llamado `invertirCadena` que tome una cadena como parámetro y devuelva una nueva cadena como valor de retorno. La nueva cadena debe contener las mismas letras que la original, pero en orden inverso. Por ejemplo, la salida del siguiente código

```
String inversa = invertirCadena ("Allen Downey");  
System.out.println (inversa);
```

debería ser

yenwoD nella

Ejercicio 5.9

- a. Creá un nuevo programa llamado Suma.java y escribí en él los siguientes métodos:

```
public static int metUno (int m, int n) {  
    if (m == n) {  
        return n;  
    } else {  
        return m + metUno (m+1, n);  
    }  
}  
  
public static int metDos (int m, int n) {  
    if (m == n) {  
        return n;  
    } else {  
        return n * metDos (m, n-1);  
    }  
}
```

- b. Escribí unas líneas en el main para testear estos métodos. Invocalos un par de veces, con distintos valores, y mirá lo que obtenés. Combinando testeo y examinación del código, descubrí lo que hacen estos métodos y dale nombres más representativos. Agregá comentarios que describan resumidamente su función.
- c. Agregá una sentencia println al principio de cada método de manera tal que se impriman los argumentos cada vez que se los invoque. Esta es una técnica muy útil para depurar programas recursivos ya que muestra el flujo de ejecución.

Ejercicio 5.10

Escribí un método recursivo llamado potencia que tome un double x y un entero n y que devuelva x^n . Ayuda: una función recursiva para esta operación es potencia (x, n) = $x * \text{potencia}(x, n-1)$. Recordá también que cualquier cosa elevado a la cero es 1.

Ejercicio 5.11

(Este ejercicio está basado en la página 44 de *Estructura e Interpretación de Programas de Computación* de Ableson y Sussman.)

El siguiente algoritmo es conocido como el algoritmo de Euclides ya que aparece en los *Elementos* de Euclides (Libro 7, año 300 a.c.). Es probablemente el algoritmo no trivial más antiguo que se conoce.

El algoritmo se basa en la observación de que, si r es el resto de dividir a por b , entonces los divisores en común entre a y b son los mismos que los divisores en común entre b y r . Así, podemos usar la ecuación

$$\text{mcd}(a, b) = \text{mcd}(b, r)$$

para reducir reiteradamente el problema de calcular un MCD al problema de calcular el MCD de pares de enteros cada vez más pequeños. Por ejemplo:

$$\text{mcd}(36, 20) = \text{mcd}(20, 16) = \text{mcd}(16, 4) = \text{mcd}(4, 0) = 4$$

implica que el MCD entre 36 y 20 es 4. Se puede demostrar que para cualquier par de números iniciales, esta reducción repetida eventualmente genera un par en el cual el segundo número es 0. Así, el MCD es el otro número del par.

Escribí un método llamado `mcd` que tome dos parámetros enteros y use el algoritmo de Euclides para calcular y devolver el máximo común divisor entre ellos.

Capítulo 6

Iteración

6.1 Asignación múltiple

No dije mucho acerca de esto, pero en Java es válido hacer más de una asignación a la misma variable. El efecto de la segunda asignación es el de reemplazar el viejo valor de la variable con uno nuevo.

```
int pepe = 5;  
System.out.print (pepe);  
pepe = 7;  
System.out.println (pepe);
```

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

Este tipo de **asignación múltiple** es la razón por la cual yo describí a las variables como *contenedores* de valores. Cuando asignás un valor a una variable, cambiás el contenido del contenedor, tal como se muestra en la figura:

<code>int pepe = 5;</code>	pepe	<div>5</div>
<code>pepe = 7;</code>	pepe	<div>5 7</div>

Cuando hay múltiples asignaciones a una variable, es especialmente importante distinguir entre una sentencia de asignación y una sentencia de comparación de igualdad. Dado que Java usa el símbolo = para la asignación, es tentador interpretar una sentencia del estilo `a = b` como si fuese una comparación de igualdad entre `a` y `b`. ¡Pero no lo es!

Para empezar, la comparación es conmutativa y la asignación no. Por ejemplo, en matemática, si $a = 7$ entonces $7 = a$. Pero en Java, $a = 7$; es una sentencia válida de asignación y $7 = a$; no lo es.

Además, en matemática, una sentencia de comparación es cierta en cualquier momento. Si $a = b$ ahora, entonces a será siempre igual a b . En Java, una asignación puede hacer que dos variables sean iguales, ¡pero no tienen que quedarse así para siempre!

```
int a = 5;
int b = a;    // a y b son ahora iguales
a = 3;        // a y b dejaron de ser iguales
```

La tercera línea cambia el valor de a pero no cambia el valor de b , y por lo tanto dejan de ser iguales. En muchos lenguajes de programación se usa un símbolo alternativo para las asignaciones, algo como \leftarrow o $:=$, para evitar esta confusión.

Si bien la asignación múltiple es usualmente útil, deberías usarla con cuidado. Si los valores de las variables están constantemente cambiando en distintas partes del programa, puede hacer que el código sea difícil de leer y de depurar.

6.2 Iteración

Una de las cosas para las cuales suelen usarse las computadoras es para la automatización de tareas repetitivas. Repetir tareas idénticas o similares sin cometer errores es algo que las computadoras hacen muy bien y las personas no.

Ya vimos programas que usan recursión para hacer repeticiones, tales como `nLineas` y `cuentaRegresiva`. Este tipo de repetición se llama **iteración**, y Java provee algunas herramientas del lenguaje que simplifican la escritura de programas iterativos.

Las dos herramientas que vamos a ver son las sentencias `while` y `for`.

6.3 La sentencia `while`

Usando una sentencia `while`, podemos reescribir `cuentaRegresiva` así:

```
public static void cuentaRegresiva (int n) {
    while (n > 0) {
        System.out.println (n);
        n = n-1;
    }
    System.out.println ("¡Explosión!");
}
```

Casi se puede leer una sentencia `while` como si estuviese en castellano¹. Lo que significa esto es, “Mientras `n` es mayor que cero, continuar imprimiendo el valor de `n` y reduciendo el valor de `n` en 1. Cuando llegues a cero, imprimir la palabra ‘¡Explosión!’”.

Más formalmente, el flujo de ejecución para una sentencia `while` es como sigue:

1. Evaluar la condición entre paréntesis produciendo `true` o `false`.
2. Si la condición es falsa, salir de la sentencia `while` y continuar la ejecución en la siguiente sentencia.
3. Si la condición es verdadera, ejecutar cada una de las sentencias que están entre las llaves y luego volver al paso 1.

A este tipo de flujos se los llama **ciclos** porque el tercer paso arma un ciclo al volver al paso 1. Notá que si la condición es falsa la primera vez, las sentencias dentro del ciclo no se ejecutan nunca. Las sentencias dentro del ciclo son usualmente llamadas el **cuerpo** del ciclo.

El cuerpo del ciclo debe cambiar el valor de una o más variables para que, eventualmente, la condición se haga falsa y el ciclo termine. Si no, el ciclo se repetirá por siempre, lo cual es conocido como un ciclo **infinito**. Una interminable fuente de diversión para los científicos de la computación es la observación de que las instrucciones en un *shampoo*, “Enjabone, enjuague, repita”, son un ciclo infinito.

En el caso de `cuentaRegresiva`, podemos probar que el ciclo terminará ya que sabemos que el valor de `n` es finito, y podemos ver que el valor de `n` toma valores cada vez más pequeños cada vez que pasa por el ciclo (en cada **iteración**), con lo cual eventualmente llegaremos a cero. En otros casos no es tan sencillo de ver:

```
public static void secuencia (int n) {
    while (n != 1) {
        System.out.println (n);
        if (n%2 == 0) {           // n es par
            n = n / 2;
        } else {                  // n es impar
            n = n*3 + 1;
        }
    }
}
```

1. N.d.T.: La palabra *while* significa *mientras* en castellano.

La condición de este ciclo es $n \neq 1$, así que el ciclo continuará hasta que n sea 1, lo que hará falsa la condición.

En cada iteración, el programa imprime el valor de n y luego verifica si es par o impar. Si es par, el valor de n se divide por dos. Si es impar, el valor se reemplaza por $3n+1$. Por ejemplo, si el valor inicial (el argumento pasado a secuencia) es 3, entonces la secuencia resultante es 3, 10, 5, 16, 8, 4, 2, 1.

Como n a veces aumenta y a veces disminuye, no hay una demostración obvia de que n alcance alguna vez 1, o de que el programa vaya a terminar. Para algunos valores particulares de n , podemos probar que termina. Por ejemplo, si el valor inicial es una potencia de dos, entonces el valor de n será par cada vez que pase por el ciclo hasta llegar a valer 1. El ejemplo anterior termina con una secuencia que empieza en 16.

Dejando de lado los valores particulares, la pregunta interesante es si podemos demostrar que este programa termina para *todo* valor de n . Hasta ahora, nadie fue capaz de demostrarlo ¡o de refutarlo!

6.4 Tablas

Una de las cosas para las cuales son buenos los ciclos es para generar e imprimir datos tabulados. Por ejemplo, antes de que las computadoras estén ampliamente disponibles, la gente tenía que calcular logaritmos, senos, cosenos, y otras funciones matemáticas comunes a mano.

Para facilitar esta tarea, había libros que contenían largas tablas en las que uno podía encontrar los valores de varias funciones. Armar las tablas era lento y aburrido, y el resultado tendía a estar lleno de errores.

Cuando las computadoras aparecieron en escena, una de las reacciones iniciales fue “¡Esto es genial! Podemos usar las computadoras para generar las tablas, y que no haya errores”. Esto resultó ser cierto (mayormente), pero bastante poco previsor. Poco tiempo después las computadoras (y calculadoras) fueron tan penetrantes que las tablas quedaron obsoletas.

Bueno, casi. Resulta que para algunas operaciones, las computadoras usan tablas de valores para obtener una respuesta aproximada, y luego realizar cálculos para mejorar la aproximación. En algunos casos, ha habido errores en las tablas subyacentes, siendo el caso más famoso el de la tabla que el Intel Pentium original usaba para realizar divisiones de punto flotante.

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

```
double x = 1.0;
while (x < 10.0) {
    System.out.println (x + " " + Math.log(x));
    x = x + 1.0;
}
```

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  1.791759469228055
7.0  1.9459101490553132
8.0  2.0794415416798357
9.0  2.1972245773362196
```

Mirando estos valores, ¿podés decir qué base usa la función log?

Dado que las potencias de dos son tan importantes en las ciencias de la computación, solemos querer encontrar los logaritmos con respecto a la base 2. Para eso, tenemos que usar la siguiente fórmula:

$$\log_2 x = \log_e x / \log_e 2 \quad (6.1)$$

Modificando la sentencia de impresión a

```
System.out.println (x + " " + Math.log(x) / Math.log(2.0));
```

se obtiene

```
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 2, ya que sus logaritmos son números enteros. Si quisiéramos hallar el logaritmo de otras potencias de dos, podríamos modificar el programa de esta manera:


```
double x = 1.0;
while (x < 100.0) {
    System.out.println (x + "    " + Math.log(x) / Math.log(2.0));
    x = x * 2.0;
}
```

Ahora, en lugar de sumar algo a x en cada iteración del ciclo, lo que da una sucesión aritmética, multiplicamos a x por algo, obteniendo así una sucesión **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 pueden haber dejado de ser útiles pero, para los científicos de la computación, ¡conocer las potencias de dos lo sigue siendo! En algún momento en que tengas un poco de tiempo libre, deberías memorizar las potencias de 2 hasta el 65536 (el cual es 2^{16}).

6.5 Tablas de dos dimensiones

Una tabla de dos dimensiones es una tabla en la que uno elige una fila y una columna y lee el valor en la intersección. Una tabla de multiplicación es un buen ejemplo. Supongamos que queremos imprimir una tabla de multiplicación para los valores de 1 a 6.

Una buena forma de empezar es escribir un ciclo simple que imprima los múltiplos de 2 todos en una línea.

```
int i = 1;
while (i <= 6) {
    System.out.print (2*i + "    ");
    i = i + 1;
}
System.out.println ("");
```

La primera línea inicializa una variable llamada i, la cual actuará como un contador, o un **iterador**. A medida que se ejecuta el ciclo, el valor de i aumenta desde 1 hasta 6, y luego cuando i vale 7, el ciclo termina. En cada iteración, imprimimos el valor 2*i seguido de tres espacios. Dado que estamos usando el comando print y no println, toda la salida aparece en una sola línea.

Como dije en la Sección 2.4, en algunos entornos la salida de `print` se almacena sin ser mostrada hasta que se invoque a `println`. Si el programa termina y te olvidaste de llamar a `println`, tal vez nunca veas la salida almacenada.

La salida de este programa es:

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

Hasta acá todo bien. El siguiente paso es **encapsular** y **generalizar**.

6.6 Encapsulamiento y generalización

Encapsular usualmente significa tomar un fragmento de código y envolverlo en un método, permitiéndote sacar ventaja de todo aquello para lo cual los métodos sirven. Ya vimos dos ejemplos de encapsulamiento cuando escribimos `imprimirParidad` en la Sección 4.3 y `esDigitoSimple` en la Sección 5.7.

Generalizar significa tomar algo específico, como imprimir múltiplos de 2, y transformarlo en algo más genérico, como imprimir múltiplos de cualquier entero.

Acá hay un método que encapsula el ciclo de la sección anterior y lo generaliza para que imprima múltiplos de cualquier entero `n`.

```
public static void imprimirMultiplos (int n) {  
    int i = 1;  
    while (i <= 6) {  
        System.out.print (n*i + "  ");  
        i = i + 1;  
    }  
    System.out.println ("");  
}
```

Para encapsular, todo lo que tuve que hacer fue agregar la primera línea, la cual declara el nombre, parámetros y tipo de retorno. Para generalizar, todo lo que tuve que hacer fue reemplazar el valor 2 con el parámetro `n`.

Si invoco 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
```

A esta altura probablemente adivines cómo vamos a imprimir una tabla de multiplicación: llamaremos a `imprimirMultiplos` repetidamente con argumentos diferentes. De hecho, vamos a usar otro ciclo para iterar por las filas.

```
int i = 1;
while (i <= 6) {
    imprimirMultiplos (i);
    i = i + 1;
}
```

Antes que nada, notá qué parecidos son este ciclo y el que está dentro de `imprimirMultiplos`. Todo lo que hice fue reemplazar la sentencia de impresión por una 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

lo cual es una tabla de multiplicación (un tanto desprolija). Si la desprolijidad te molesta, Java provee métodos que te dan más control sobre el formato de salida, pero no voy a ahondar en eso ahora.

6.7 Métodos

En la última sección mencioné “todas las cosas para las que sirven los métodos.” A esta altura, te estarás preguntando qué son esas cosas. Aquí algunas de las razones por las que son útiles:

- Al darle nombre a una secuencia de sentencias, hacés tu programa más fácil de leer y de depurar.
- Dividir un programa largo en métodos te permite separar las partes del programa, depurarlas, y luego componerlas como una unidad.
- Los métodos hacen más fácil o permiten la iteración y la recursión.
- Los métodos bien diseñados son útiles para más de un programa. Una vez que escribís uno y te asegurás que esté libre de errores, lo podés usar en varios programas.

6.8 Más encapsulamiento

Para mostrar el encapsulamiento nuevamente, tomaré el código de la sección anterior y lo encapsularé en un método:

```
public static void imprimirTablaMultiplos () {  
    int i = 1;  
    while (i <= 6) {  
        imprimirMultiplos (i);  
        i = i + 1;  
    }  
}
```

El proceso que estoy mostrando es un plan de desarrollo muy común. Desarrollás código gradualmente, agregando líneas al main, o algún otro lado, y cuando lo tenés funcionando, lo extraés y lo encapsulás en un método.

La razón por la que esto es útil es que a veces no sabés cuando comenzás a escribir exactamente cómo dividir el programa en métodos. Este enfoque te permite diseñar a medida que vas avanzando.

6.9 Variables locales

Probablemente te estés preguntando cómo podemos usar la misma variable *i* tanto en `imprimirMultiplos` como en `imprimirTablaMultiplos`. ¿No he dicho que sólo se puede declarar una variable una sola vez? ¿Y no causa problemas cuando uno de los métodos cambia el valor de la variable?

La respuesta a ambas preguntas es “no”, porque la variable *i* usada en `imprimirMultiplos` y la variable *i* usada en `imprimirTablaMultiplos` *no son la misma variable*. Tienen el mismo nombre, pero no hacen referencia a la misma ubicación de almacenamiento, y cambiar el valor de una de ellas no tiene ningún efecto sobre el valor de la otra.

Las variables que se declaran dentro de la definición de un método se llaman **variables locales** porque son locales a sus propios métodos. No podés acceder a una variable local desde afuera de su método “padre”, y podés crear tantas variables con el mismo nombre, siempre y cuando no estén en el mismo método.

A menudo es una buena idea utilizar nombres de variables distintos en métodos distintos, para evitar confusión, pero hay buenas razones para reutilizar nombres. Por ejemplo, es común usar los nombres *i*, *j* y *k* como variables para ciclos. Si evitás usarlas porque ya las usaste en otro lado, probablemente harás al programa más difícil de leer.

6.10 Más generalización

Como otro ejemplo de generalización, imaginá que querías un programa que imprimiera una tabla de multiplicación de cualquier tamaño, no sólo de 6x6. Podrías añadir un parámetro a `imprimirTablaMultiplos`:

```
public static void imprimirTablaMultiplos (int largo) {  
    int i = 1;  
    while (i <= largo) {  
        imprimirMultiplos (i);  
        i = i + 1;  
    }  
}
```

Reemplacé el valor 6 con el parámetro `largo`. De esta manera, si llamo a `imprimirTablaMultiplos` 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

que está bien, excepto que probablemente quiera que la tabla sea cuadrada (igual número de filas que de columnas), lo cual implica que tengo que añadir otro parámetro a `imprimirMultiplos`, para especificar cuántas columnas debe tener la tabla.

Sólo para molestar, voy a llamar a ese parámetro `largo`, demostrando que métodos distintos pueden tener parámetros con el mismo nombre (tal como ocurre con las variables locales):

```
public static void imprimirMultiplos (int n, int largo) {  
    int i = 1;  
    while (i <= largo) {  
        System.out.print (n*i + "    ");  
        i = i + 1;  
    }  
    nuevaLinea();  
}
```

```

public static void imprimirTablaMultiplos (int largo) {
    int i = 1;
    while (i <= largo) {
        imprimirMultiplos (i, largo);
        i = i + 1;
    }
}

```

Notar que cuando agregué un nuevo parámetro, tuve que cambiar la primera línea del método (la interface o prototipo), y también tuve que cambiar el lugar donde se llamaba al método en `imprimirTablaMultiplos`. Como se esperaba, 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 generalizás un método de forma apropiada, a menudo encontrás que el programa resultante tiene capacidades que no pretendías darle. Por ejemplo, podés notar que la tabla resultante es simétrica, porque $a \times b = b \times a$, con lo cual todos los elementos en la tabla aparecen por duplicado. Podrías ahorrar tinta imprimiendo sólo la mitad de la tabla. Para hacer eso sólo tendrías que cambiar una línea en `imprimirTablaMultiplos`. Cambiar

```

    imprimirMultiplos (i, largo);

```

a

```

    imprimirMultiplos (i, i);

```

y obtenés

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 dejo a vos que deduzcas cómo funciona.

6.11 Glosario

ciclo: Sentencia que se ejecuta repetidamente mientras o hasta cierta condición se satisfaga.

ciclo infinito: Ciclo cuya condición siempre es verdadera.

cuerpo: Sentencias dentro del ciclo.

iteración: “Pasada” (ejecución) del cuerpo del ciclo, incluyendo la evaluación de la condición.

encapsular: Dividir un programa largo y complejo en componentes (como métodos) y aislar los componentes entre sí (por ejemplo, a través del uso de variables locales).

variable local: Variable que se declara dentro de un método y que existe sólo dentro de ese método. Las variables locales no pueden ser accedidas desde afuera del método padre y no interfieren con ningún otro método.

generalizar: Reemplazar algo innecesariamente específico (como un valor constante) con algo apropiadamente general (como una variable o un parámetro). La generalización hace más versátil al código, más fácil de reutilizar, y a veces incluso más fácil de escribir.

plan de desarrollo: Proceso para desarrollar un programa. En este capítulo, demostré un estilo de desarrollo basado en escribir código para hacer cosas simples y específicas, y luego encapsularlo y generalizar. En la Sección 5.2 mostré una técnica que llamé desarrollo incremental. En capítulos posteriores mostraré otros estilos de desarrollo.

6.12 Ejercicios

Ejercicio 6.1

```
public static void main (String[] args) {  
    ciclar (10);  
}  
  
public static void ciclar(int n) {  
    int i = n;  
    while (i > 0) {  
        System.out.println (i);
```

```

        if (i%2 == 0) {
            i = i/2;
        } else {
            i = i+1;
        }
    }
}

```

- Escribí una tabla que muestre los valores de las variables *i* y *n* durante la ejecución de *ciclar*. La tabla debe contener una columna para cada variable y una línea para cada iteración.
- ¿Cuál es la salida del programa?

Ejercicio 6.2

- Encapsular el siguiente fragmento de código, transformándolo en un método que toma un *String* como parámetro y que devuelve (y no imprime) el resultado final de *cont*.
- En una oración, describí en forma abstracta qué es lo que hace el método.
- Asumiendo que ya generalizaste este método de modo que funciona en cualquier *String*, ¿qué más podés hacer para generalizarlo aún más?

```

String s = "((3 + 7) * 2)";
int largo = s.length ();

```

```

int i = 0;
int cont = 0;

```

```

while (i < largo) {
    char c = s.charAt(i);

    if (c == '(') {
        cont = cont + 1;
    } else if (c == ')') {
        cont = cont - 1;
    }
    i = i + 1;
}

```

```

System.out.println (cont);

```


Ejercicio 6.3

Digamos que te dan un número, a , y querés encontrar su raíz cuadrada. Una forma de hacer esto es comenzar con una aproximación muy burda de la respuesta, x_0 , y luego mejorar la aproximación usando la siguiente fórmula:

$$x_1 = (x_0 + a/x_0)/2$$

Por ejemplo, si queremos encontrar la raíz cuadrada de 9, y comenzamos con $x_0 = 6$, entonces $x_1 = (6 + 9/6)/2 = 15/4 = 3,75$, que está más cerca.

Podemos repetir este procedimiento, usando x_1 para calcular x_2 , y así siguiendo. En este caso, $x_2 = 3,075$ y $x_3 = 3,00091$. Con lo cual está convergiendo rápidamente a la respuesta correcta (que es 3).

Escribir un método llamado `raizCuadrada` que toma un `double` como parámetro y devuelve una aproximación de la raíz cuadrada del parámetro, usando este algoritmo. No podés utilizar el método `Math.sqrt`.

Como aproximación inicial, usá $a/2$. Tu método debe iterar hasta conseguir dos aproximaciones consecutivas que difieren en menos de 0.0001; en otras palabras, hasta que el valor absoluto de $x_n - x_{n-1}$ es menor a 0.0001. Podés usar el método `Math.abs` de la librería para calcular el valor absoluto.

Ejercicio 6.4

En el ejercicio 5.10 escribimos una versión recursiva de potencia, que toma un `double x` y un entero n y devuelve x^n . Ahora escribí una versión iterativa para hacer el mismo cálculo.

Ejercicio 6.5

La Sección 5.10 presenta una versión recursiva para calcular la función factorial. Escribí una versión iterativa de `factorial`.

Ejercicio 6.6

Una forma de calcular e^x es utilizar la expansión infinita de la serie

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \dots$$

Si la variable del ciclo se llama i , luego el i -ésimo término es igual a $x^i/i!$.

- Escribí un método llamado `miexp` toma un `double x` y un entero n y suma los primeros n términos de la serie mostrada arriba. Podés usar la función factorial de la Sección 5.10 o tu propia versión iterativa.
- Podés hacer este método mucho más eficiente si te das cuenta que en cada iteración el numerador del término es el mismo que el de su predecesor multiplicado por x y el denominador es el mismo que el de su predecesor multiplicado por i . Usá esta observación para eliminar el uso de `Math.pow` y `factorial`, y verificá que seguís obteniendo el mismo resultado.

- c. Escribí un método llamado `verificar` que toma un solo parámetro, `x`, e imprime los valores de `x`, `Math.exp(x)` y `miexp(x, n)` para algún valor de `n`. La salida debe verse parecida a:

```
1.0      2.718281828459045      2.708333333333333
```

PISTA: podés usar la cadena `"\t"` para imprimir un carácter de tabulación entre las columnas de una tabla.

- d. Variar la cantidad de términos usados en la serie (el segundo argumento que `verificar` le pasa a `miexp`) y ver el efecto en la precisión del resultado. Ajustar este parámetro hasta que el valor se corresponda con la respuesta “correcta” cuando `x` es 1.
- e. Escribir un ciclo en `main` que llama a `verificar` con los valores 0.1, 1.0, 10.0, y 100.0. ¿Cómo varía la precisión del resultado a medida que varía `x`? Comparar el número de dígitos en que coinciden en lugar de la diferencia entre los valores.
- f. Agregar un ciclo en `main` que prueba `miexp` con los valores -0.1, -1.0, -10.0, y -100.0. Comentar sobre la precisión.

Ejercicio 6.7

Una forma de evaluar e^{-x^2} es utilizar la expansión infinita de la serie

$$e^{-x^2} = 1 + 2x + 3x^2/2! + 4x^3/3! + 5x^4/4! + \dots$$

En otras palabras, necesitamos sumar los términos en los que el i -ésimo término es igual a $(i + 1)x^i/i!$. Escribí un método llamado `gauss` que toma `x` y `n` como parámetros y devuelve la suma de los primeros `n` términos de la serie. No podés utilizar `factorial` ni potencia.

Capítulo 7

Cadenas y cosas

7.1 Invocando métodos en objetos

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

De los tipos que hemos visto hasta ahora, sólo los `String` son objetos. Basado en la definición de objeto, te podés llegar a preguntar “¿Cuáles son los datos contenidos en un objeto de tipo `String`?” y “¿Cuáles son los métodos que podemos llamar en objetos de tipo `String`?”

Los datos contenidos en un objeto `String` son las letras de la cadena. Hay bastantes métodos que operan sobre `Strings`, pero usaré sólo unos pocos en este libro. El resto están documentados (en inglés) en

<http://java.sun.com/j2se/1.4.1/docs/api/java/lang/String.html>

El primer método que veremos es `charAt`, que te permite extraer letras de un `String`. Para poder almacenar el resultado, necesitamos un tipo de variable capaz de almacenar letras individuales (a diferencia de las cadenas). Las letras individuales se llaman caracteres, y el tipo de variable que los almacena es `char`.

Los `chars` funcionan igual que los otros tipos de datos que vimos:

```
char pepe = 'c';
if (pepe == 'c') {
    System.out.println (pepe);
}
```

Los valores que simbolizan caracteres se escriben entre comillas simples ('c'). A diferencia de las cadenas (que se escriben con comillas dobles), los caracteres pueden contener una sola letra o símbolo. Así es cómo se utiliza el método `charAt`:

```
String fruta = "banana";
char letra = fruta.charAt(1);
System.out.println (letra);
```

La sintaxis `fruta.charAt` indica que estoy llamando el método `charAt` en el objeto llamado `fruta`. Estoy pasando el parámetro 1 a este método, que indica que quisiera obtener la primera letra de la cadena. El resultado es un carácter, el cual se almacena en una variable `char` llamada `letra`. Cuando imprimo el valor de `letra`, me encuentro con una sorpresa:

a

a no es la primera letra de "banana". A menos que seas un científico de la computación. Por razones perversas, los científicos de la computación empiezan siempre a contar de cero. La 0-ésima letra ("cero-ésima") de "banana" es b. La 1-ésima ("un-ésima") es a y la 2-ésima ("dos-ésima") letra es n. Si querés la cero-ésima letra de una cadena, tenés que pasar 0 como parámetro:

```
char letra = fruta.charAt(0);
```

7.2 Largo

El segundo método de `String` que veremos es `length`, el cual devuelve el número de caracteres de la cadena. Por ejemplo:

```
int largo = fruta.length();
```

`length` no toma ningún parámetro, tal como lo indica `()`, y devuelve un entero, en este caso 6.

Para conseguir la última letra de una cadena, podrías estar tentado de hacer algo como lo siguiente:

```
int largo = fruta.length();
char ultimo = fruta.charAt (largo);           // MAL!!
```

Eso no va a funcionar. La razón es que no hay una 6-ésima letra en la cadena "banana". Dado que comenzamos contando desde 0, las 6 letras se numeran del 0 al 5. Para obtener el último carácter, se tiene que restar 1 a `largo`.

```
int largo = fruta.length();
char ultimo = fruta.charAt (largo-1);
```

7.3 Recorrido

Algo común para hacer con una cadena es comenzar desde el principio, seleccionar cada carácter por vez, hacer algo con él, y continuar hasta el fin. Este patrón de procesamiento se llama **recorrido**. Una forma natural de codificar un recorrido es mediante una sentencia `while`:

```
int indice = 0;
while (indice < fruta.length()) {
    char letra = fruta.charAt (indice);
    System.out.println (letra);
    indice = indice + 1;
}
```

Este ciclo recorre la cadena e imprime cada letra en líneas separadas. Notá que la condición del ciclo es `indice < fruta.length()`, lo cual quiere decir que cuando `indice` es igual al largo de la cadena, la condición es falsa y el cuerpo del ciclo no se ejecuta. El último carácter al que accedemos es aquel con índice `fruta.length()-1`.

El nombre de la variable del ciclo es `indice`. Un **índice** es una variable o valor usado para especificar un miembro de un conjunto ordenado (en este caso el conjunto de caracteres en la cadena). El índice indica (de aquí el nombre) a cuál querés hacer referencia. El conjunto debe ser ordenado¹ de modo que cada carácter tenga un índice y cada índice tenga un carácter.

Como ejercicio, escribí un método que tome un `String` como parámetro y que imprima las letras al revés sin saltar de línea.

7.4 Errores en tiempo de ejecución

Hace tiempo, 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 comenzado a ejecutarse. En Java, los errores en tiempo de ejecución se llaman **excepciones**.

1. N.d.T.: Con ordenado no se refiere a que los elementos tienen que estar ordenados de menor a mayor, sino a que siguen un orden particular y que el conjunto de caracteres “arma” no es el mismo que “amar”, a pesar de que tengan los mismos elementos.

Hasta ahora, probablemente no has visto muchos errores en tiempo de ejecución, porque no hemos estado haciendo muchas cosas que pudieran causar uno. Bueno, ahora sí. Si usás el método `charAt` y le pasás un índice que es negativo, o mayor a `length-1`, obtendrás una excepción: en particular, una `StringIndexOutOfBoundsException`². Probálo y fijáte cómo se muestra.

Si tu programa produce una excepción, imprime un mensaje de error indicando el tipo de excepción y en qué parte del programa se desató. Luego, el programa finaliza.

7.5 Leyendo la documentación

Si vas a la siguiente dirección en internet

<http://java.sun.com/j2se/1.4/docs/api/java/lang/String.html>

y hacés click en `charAt`, vas a obtener la siguiente documentación³ (o algo parecido):

```
public char charAt(int index)
```

Returns the character at the specified index.

An index ranges from 0 to `length() - 1`.

Parameters: `index` - the index of the character.

Returns: the character at the specified index of this string.

The first character is at index 0.

Throws: `StringIndexOutOfBoundsException` if the index is out of range.

La primera línea es el **prototipo** del método, que indica el nombre del método, el tipo de sus parámetros, y el tipo que devuelve.

2. N.d.T.: “índice fuera de rango”.

3. N.d.T.: Lamentablemente, la documentación oficial de Java se encuentra en inglés. Probablemente podés encontrar alguna traducción en Internet por vías alternativas, pero la documentación oficial viene sólo en este idioma. Si el inglés es un problema, una posible solución es valerse de algún traductor automático (como Google Translate), pero con muchísimo cuidado porque el traductor probablemente también traduzca nombres de métodos u otras palabras del lenguaje que no se pueden modificar.

La siguiente línea describe lo que hace el método. A continuación se explican los parámetros y los valores que devuelve. En este caso las explicaciones son un poco redundantes, pero la documentación intenta adecuarse a un formato estándar. Finalmente explica qué excepciones, en caso de que haya alguna, pueden ser provocadas por este método.

7.6 El método `indexOf`

En algún sentido, `indexOf` es el opuesto de `charAt`. `charAt` toma un índice y devuelve el carácter en dicha posición. `indexOf` toma un carácter y encuentra el índice donde aparece ese carácter.

El método `charAt` falla si el índice está fuera de rango, y causa una excepción. `indexOf` falla si el carácter no aparece en la cadena, y devuelve el valor `-1`.

```
String fruta = "banana";  
int indice = fruta.indexOf('a');
```

Esto busca el índice de la letra `'a'` en la cadena. En este caso, la letra aparece tres veces, con lo cual no es obvio lo que `indexOf` debería hacer. Según la documentación, devuelve el índice de su *primera* aparición.

Para obtener apariciones posteriores, hay una versión alternativa de `indexOf` (para una explicación sobre métodos con distintos parámetros, ver la Sección 5.4). Toma un segundo parámetro que indica desde qué parte de la cadena debe comenzar a buscar. Si llamamos

```
int indice = fruta.indexOf('a', 2);
```

comenzará a buscar desde la 2-ésima letra (la primera `n`) y encontrará la segunda `a`, que se encuentra en el índice 3. Si la letra buscada aparece en el índice de inicio, entonces el índice de inicio es la respuesta. Luego,

```
int indice = fruta.indexOf('a', 5);
```

devuelve 5. Basados en la documentación, es un poco engañoso entender qué ocurre si el índice de inicio está fuera de rango:

`indexOf` devuelve el índice de la primera aparición del carácter en la secuencia de caracteres representada por este objeto tal que es mayor o igual que `indiceInicio`, o `-1` si el carácter no aparece.

Una forma de entender qué quiere decir esto es probar con un par de casos. Aquí están los resultados de mis experimentos:

- Si el índice de inicio es mayor o igual que el largo (`length()`), el resultado es `-1`, indicando que la letra no aparece en ningún índice mayor que el índice de inicio.
- Si el índice de inicio es negativo, el resultado es `1`, indicando la primera aparición de la letra en un índice mayor que el índice de inicio.

Si volvéis atrás y leéis la documentación, verás que este comportamiento es consistente con la definición, incluso si no era obvio desde un principio. Ahora que tenemos una mejor idea de cómo funciona `indexOf`, podemos utilizarlo como parte de un programa.

7.7 Iterando y contando

El siguiente programa cuenta la cantidad de veces que aparece la letra 'a' en la cadena:

```
String fruta = "banana";
int largo = fruta.length();
int contador = 0;

int indice = 0;
while (indice < largo) {
    if (fruta.charAt(indice) == 'a') {
        contador = contador + 1;
    }
    indice = indice + 1;
}
System.out.println (contador);
```

Este programa demuestra un elemento común, llamado **contador**. La variable `contador` se inicializa en `0` y luego se incrementa cada vez que encontramos una 'a' (**incrementar** es aumentar en uno; su opuesto es **decrementar**). Cuando salimos del ciclo, `contador` contiene el resultado: la cantidad total de a's.

Como ejercicio, encapsulá este código en un método llamado, por ejemplo, `contarLetras` y generalizalo para que reciba la cadena y la letra como argumentos.

Como otro ejercicio, reescribí este método de modo que use `indexOf` para buscar las letras, en lugar de mirar cada carácter uno por uno.

7.8 Operadores de incremento y decremento

Incrementar y decrementar son operaciones tan comunes que Java provee operadores especiales para ellas. El operador `++` aumenta en uno

el valor de un `int` o un `char`. -- resta uno. Ninguno de los dos operadores funciona en `double`, `boolean` o `String`.

Técnicamente, es válido incrementar una variable y utilizarla en una expresión al mismo tiempo. Por ejemplo, podés ver algo como:

```
System.out.println (i++);
```

Al ver esto, no queda claro si el incremento ocurrirá antes o después de que el valor sea impreso. Dado que expresiones como esta tienden a ser confusas, no te recomendaría usarlas. De hecho, para disuadirte aún más, no te voy a decir cuál es el resultado. Si realmente querés saberlo, podés probarlo.

Usando los operadores de incremento, podemos reescribir el contador de letras:

```
int indice = 0;
while (indice < largo) {
    if (fruta.charAt(indice) == 'a') {
        contador++;
    }
    indice++;
}
```

Es un error común escribir algo como:

```
indice = indice++;           // MAL!!
```

Lamentablemente, esto es sintácticamente correcto, con lo que el compilador no te lo advertirá. El efecto de esta sentencia es la de dejar el valor de `indice` sin modificar. Este es un error que, a menudo, es difícil de encontrar.

Acordate que podés escribir “`indice = indice+1;`” o podés escribir “`indice++;`”, pero no deberías mezclarlos.

7.9 Los Strings son inmutables

Si mirás la documentación de los métodos de `String`, podés llegar a notar los métodos `toUpperCase`⁴ and `toLowerCase`⁵. Estos métodos son, a menudo, una fuente de confusión, porque suena como si tuvieran el efecto de cambiar(o mutar) una cadena preexistente. En realidad, ninguno de estos métodos, ni ningún otro puede modificar una cadena, porque la cadenas son **inmutables**.

4. N.d.T.: aMayúsculas.

5. N.d.T.: aMinúsculas.

Cuando llamás a `toUpperCase` en un `String`, te devuelve un *nuevo* `String`. Por ejemplo:

```
String nombre = "Alan Turing";
String nombreMayusculas = nombre.toUpperCase ();
```

Después de que la segunda línea se ejecuta, `nombreMayusculas` contiene el valor `"ALAN TURING"`, pero `nombre` aún contiene `"Alan Turing"`.

7.10 Los `Strings` no son comparables

A menudo, es necesario comparar cadenas para saber si es la misma, o para ver cuál viene antes en orden alfabético. Sería cómodo poder utilizar los operadores de comparación, como `==` y `>`, pero no podemos.

Para poder comparar `Strings`, tenemos que usar los métodos `equals` y `compareTo`. Por ejemplo:

```
String nombre1 = "Alan Turing";
String nombre2 = "Ada Lovelace";

if (nombre1.equals (nombre2)) {
    System.out.println ("Los nombres son iguales.");
}

int bandera = nombre1.compareTo (nombre2);
if (bandera == 0) {
    System.out.println ("Los nombres son iguales.");
} else if (bandera < 0) {
    System.out.println ("nombre1 viene antes que nombre2.");
} else if (bandera > 0) {
    System.out.println ("nombre2 viene antes que nombre1.");
}
```

La sintaxis acá es un poco extraña. Para comparar dos cosas, tenés que llamar a un método en una de ellas y pasar la otra como parámetro.

El valor que devuelve `equals` es suficientemente claro; `true` si las cadenas contienen los mismos caracteres (en el mismo orden), y `false` si no.

El valor que devuelve `compareTo` es un poco raro. Es la diferencia entre el primer par de caracteres de las cadenas que son distintos. Si las cadenas son iguales, es 0. Si la primera cadena (aquella sobre la que se llama el método) viene primero en el alfabeto, la diferencia es negativa. De lo

contrario, la diferencia es positiva. En el caso del ejemplo, el valor que devuelve es 8 positivo, porque la segunda letra de “Ada” viene antes que la segunda letra de “Alan” por 8 letras.

Usar `compareTo` es muchas veces engañoso, y nunca recuerdo la forma exacta de usarlo, pero las buenas noticias es que esta interfaz es bastante estándar para comparar muchos tipos de objetos, así que cuando lo aprendas para uno, lo aprendiste para todos.

Sólo para dejar esta sección completa, debo admitir que es *válido*, pero muy rara vez *correcto*, usar el operador `==` con `Strings`. Sin embargo, su significado no cobrará sentido hasta más tarde, por lo que, por ahora, no lo uses.

7.11 Glosario

objeto: Colección de datos relacionados, que vienen con un conjunto de métodos que operan sobre sí misma. Los objetos que hemos usado hasta ahora sólo son los del tipo `String`.

índice: Variable o valor usada para seleccionar un miembro de un conjunto ordenado, como un carácter de una cadena.

recorrer: Iterar a través de todos los elementos de un conjunto efectuando una operación similar en cada uno de ellos.

contador: Variable utilizada para contar algo, generalmente inicializada en 0 y luego incrementada.

incrementar: Aumentar el valor de una variable en uno. El operador de incremento en Java es `++`.

decrementar: Decrementar el valor de una variable en uno. El operador de decremento en Java es `--`.

excepción: Error en tiempo de ejecución. Las excepciones provocan que el programa termine su ejecución.

7.12 Ejercicios

Ejercicio 7.1

El objetivo de este ejercicio es probar algunas de las operaciones de `String` y profundizar en algunos detalles que no fueron cubiertos en este capítulo.

- Crear un nuevo programa llamado `Prueba.java` y escribir un `main` que contenga diversos tipos usando el operador `+`. Por ejemplo, ¿qué pasa si “sumás” un `String` y un `char`? ¿Efectúa una adición o una concatenación?

¿Cuál es el tipo del resultado? (¿Cómo podés determinar el tipo del resultado?)

- b. Hacé una copia en grande de esta tabla y llenala. En la intersección de cada par de tipos, debés indicar si es válido usar el operador + con estos tipos, qué operación se efectúa (adición o concatenación), y cuál es el tipo del resultado.

	boolean	char	int	String
boolean				
char				
int				
String				

- c. Pensá acerca de las decisiones que tomaron los diseñadores de Java cuando ellos llenaron esta tabla. ¿Cuántas de estas decisiones parecen ser la única opción razonable que había? ¿Cuántas parecen decisiones arbitrarias entre varias opciones igualmente razonables? ¿Cuántas parecen estúpidas?
- d. Un acertijo: generalmente, la sentencia `x++` es exactamente equivalente a la sentencia `x = x+1`. ¡A menos que `x` sea un `char`! En ese caso, `x++` es válido, pero `x = x+1` genera un error. Probalo y comprobá cuál es el mensaje de error, luego fijate si podés explicar qué es lo que está pasando.

Ejercicio 7.2

¿Cuál es la salida de este programa? Describí en una oración, en forma abstracta, qué es lo que hace `bing` (no cómo funciona).

```
public class Misterio {

    public static String bing (String s) {
        int i = s.length() - 1;
        String total = "";

        while (i >= 0 ) {
            char ch = s.charAt (i);
            System.out.println (i + "      " + ch);

            total = total + ch;
            i--;
        }
        return total;
    }

    public static void main (String[] args) {
        System.out.println (bing ("Allen"));
    }
}
```

Ejercicio 7.3

Un amigo tuyo te muestra el siguiente método y te explica que si `numero` es un número de dos dígitos, el programa mostrará el número al revés. Asegura que si `numero` es 17, el método imprimirá 71.

¿Tiene razón? Si no, explicá qué es lo que el programa realmente hace y modifícalo de modo que haga lo correcto.

```
int numero = 17;
int ultimoDigito = numero%10;
int primerDigito = numero/10;
System.out.println (ultimoDigito + primerDigito);
```

Ejercicio 7.4

¿Cuál es la salida del siguiente programa?

```
public class Rarificar {

    public static void rarificar (int x) {
        if (x == 0) {
            return;
        } else {
            rarificar(x/2);
        }

        System.out.print (x%2);
    }

    public static void main (String[] args) {
        rarificar(5);
        System.out.println ("");
    }
}
```

Explicar en 4 o 5 palabras qué es lo que realmente hace el método `rarificar`.

Ejercicio 7.5

Realizar las siguientes tareas:

- Crear un nuevo programa llamado `Capicua.java`.
- Escribir un método llamado `primera` que toma un `String` y devuelve la primera letra, y un llamado `ultima` que devuelve la última letra.

- c. Escribir un método llamado `medio` que toma un `String` y devuelve una subcadena que contiene todo *excepto* la primera y la última letra.
Pista: buscar información sobre el método `substring` en la clase `String`.
Hacé algunas pruebas para asegurarte que entendés bien cómo funciona `substring` antes de comenzar a escribir `medio`.
¿Qué pasa si llamás a `medio` con una cadena que tiene sólo dos letras? ¿Y una letra? ¿Y una vacía?
- d. La definición común de *capicúa* es una palabra que se lee igual al derecho que al revés, como “neuquen” y “palindromoomordnilap.” Una forma alternativa de definir esta propiedad adecuadamente es especificar una forma de comprobarla. Por ejemplo, podemos decir, “una palabra de una letra es *capicúa*, y una palabra de dos letras es *capicúa* si las letras son iguales, y cualquier otra palabra es un *palíndromo* si la primera letra es igual a la última y lo que queda en el medio es *capicúa*”.
Escribí un método recursivo `esCapicua` que toma un `String` y devuelve un booleano especificando si la palabra es *capicúa* o no.
- e. Una vez que tengas un comprobador de palabras *capicúa*, buscá la forma de simplificarlo mediante el uso de menos condiciones. Pista: puede ser útil adoptar la definición de que una cadena vacía es *capicúa*.
- f. En una hoja, pensá una estrategia para comprobar si es *capicúa* de forma iterativa. Hay varias estrategias, con lo cual, asegurate de tener un plan definido antes de comenzar a escribir código.
- g. Implementá tu estrategia en un método `esCapicuaIter`.

Ejercicio 7.6

Una palabra se dice que es “*abecedaria*” si las letras en la palabra aparecen en orden alfabético. Por ejemplo, la siguientes son todas palabras *abecedarias* del castellano.

adiós, afín, afinó, ágil, bello, celos, cenó, chinos
dijo, dimos, dios, fijos, finos, hijos, hilos, himno

- a. Describí un algoritmo para decidir si una palabra dada (`String`) es *abecedaria*, asumiendo que la palabra contiene sólo letras minúsculas. Tu algoritmo puede ser iterativo o recursivo.
- b. Implementar el algoritmo en un método `esAbecedaria`.
- c. ¿Funciona el algoritmo si le pasamos como parámetro “*ágil*”? En caso negativo, ¿por qué te parece que puede ser? ¿Cómo lo solucionarías?

Ejercicio 7.7

Decimos que una palabra es “*duódroma*” si contiene sólo letras duplicadas, como “*llaammaa*” o “*ssaabb*”. Conjeturo que no hay palabras *duódromas* en el castellano. Para corroborarlo, quisiera un programa que lea palabras del diccionario, una por vez, y verifique si son *duódromas*.

Escribir un método `esDuodroma` que toma un `String` y devuelve un boolean indicando si la palabra es o no duódroma.

Ejercicio 7.8

Cuando se graban nombres en una computadora, a veces se escriben con el nombre de pila primero, como “Allen Downey,” y a veces, con el apellido primero, seguido de una coma y el nombre de pila, por ejemplo “Downey, Allen.” Eso puede dificultar el trabajo de comparar nombres y ponerlos en orden alfabético.

Un problema relacionado es que a algunos nombres tienen letras mayúsculas en lugares raros, como mi amiga Beth DeSombre, o ninguna mayúscula, como el poeta e.e. cummings. Cuando las computadoras comparan caracteres, usualmente consideran que todas las minúsculas vienen antes que la primera de las mayúsculas. Como resultado, las computadoras ponen nombres con mayúsculas en un orden incorrecto.

Si sos curioso, podés buscar la guía telefónica y ver si podés descubrir el esquema de ordenamiento. Buscá nombres compuestos como Van Houten y nombres con mayúsculas en lugares poco comunes, como desJardins. Si tenés acceso a alguna otra guía o directorio, probá mirarla también, a ver si los esquemas difieren.

El resultado de toda esta falta de estandarización es que generalmente no es correcto ordenar nombres usando la comparación de `Strings`. Una solución común es mantener dos versiones de cada nombre: la versión imprimible y la versión interna usada para ordenamiento.

En este ejercicio, escribirás un método que compara nombres convirtiéndolos al formato estándar. Trabajaremos desde lo más particular hasta llegar a lo más abstracto⁶ escribiendo algunos métodos auxiliares para finalmente escribir `compararNombre`.

- Crear un nuevo programa llamado `Nombre.java`. En la documentación de `String`, informate sobre qué hacen `find`, `toLowerCase` y `compareTo`. Escribí algunas pruebas para asegurarte de entender cómo funcionan.
- Escribí un método llamado `tieneComa` que toma un nombre como parámetro y devuelve un booleano indicando si contiene una coma. Si la tiene, podés asumir que tiene la forma apellido seguido de coma y el nombre.
- Escribí un método llamado `convertirNombre` que toma un nombre como parámetro. Debería verificar si contiene una coma. Si tiene, entonces debe simplemente devolver ese mismo valor.

Si no, entonces debe asumir que el nombre está en el formato en que el nombre de pila está al principio, y debe devolver una nueva cadena con el

6. N.d.T.: Esta estrategia se conoce como *bottom-up* — de abajo hacia arriba —, en contraposición con *top-down* — de arriba hacia abajo — y se refieren al orden en que se va construyendo un programa, si desde lo más abstracto y se va refinando hacia lo particular, o al revés, comenzando con el detalle y se va construyendo, algo cada vez más general.

formato en que el apellido va al principio, una coma, y finalmente el nombre.

- d. Escribir un método `compararNombre` que toma dos nombres como parámetros y devuelve -1 viene alfabéticamente antes que el segundo, 0 si son iguales, y 1 si el segundo viene alfabéticamente antes. El método debe ser insensible a las mayúsculas, es decir, no debe importar si las letras son mayúsculas o minúsculas.

Ejercicio 7.9

- a. El anillo decodificador del Capitán Crunch funciona tomando cada letra en una cadena y sumándole 13. Por ejemplo, 'a' se transforma en 'n' y 'b' en 'o'. Las letras “dan la vuelta” al final, de modo que 'z' se convierte en 'm'.

Escribir un método que tome un `String` y devuelva uno nuevo conteniendo la versión codificada. Deberás asumir que la cadena contiene letras mayúsculas y minúsculas, y espacios, pero ningún signo de puntuación. Las letras minúsculas deberán ser transformadas a otras minúsculas y las mayúsculas a otras mayúsculas. Los espacios no se deben codificar.

- b. Generalizá el método del Capitán Crunch de modo que en lugar de sumarle 13 a las letras, le sume un número dado. Ahora deberías ser capaz de codificar cadenas sumándoles 13 y decodificarlas sumándoles -13. Probálo.

Capítulo 8

Objetos interesantes

8.1 ¿Qué es interesante?

A pesar de que los `Strings` son objetos, no son objetos muy interesantes, porque

- son inmutables.
- no tienen variables miembro.
- no necesitas usar el comando `new` para crear uno.

En este capítulo, vamos a tomar dos tipos de objetos que son parte del lenguaje Java, `Point`¹ y `Rectangle`². Ya desde el comienzo, quiero dejar claro que estos puntos y rectángulos no son objetos gráficos que aparecen en la pantalla. Son variables que contienen datos, tal como los `int` y los `double`. Como otras variables, son utilizadas internamente para realizar cálculos.

Las definiciones de las clases `Point` y `Rectangle` están en el paquete `java.awt`, de modo que tenemos que importarlas.

8.2 Paquetes

Las clases preincorporadas de Java se dividen en **paquetes**, incluyendo `java.lang`, que contiene la mayoría de las clases que hemos visto hasta ahora, y `java.awt`, que contiene las clases que pertenecen al **Abstract**

1. N.d.T: Punto.

2. N.d.T: Rectángulo.

Window Toolkit³ (AWT), que contiene las clases para ventanas, botones, gráficos, etc.

Para poder usar este paquete, tenés que **importarlo**, razón por la cual el programa en la Sección D.1 comienza con `import java.awt.*`. El `*` indica que queremos importar todas las clases del paquete AWT. Si querés, podés nombrar las clases que querés importar explícitamente, pero no hay un gran ventaja en hacerlo. Las clases en `java.lang` se importan automáticamente, lo cual explica que muchos de nuestros programas no hayan requerido una sentencia `import`.

Todas las sentencias `import` aparecen al inicio del programa, fuera de la definición de clase.

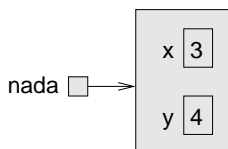
8.3 Objetos Point

Al nivel más básico, un punto son dos números (coordenadas) a las que tratamos conjuntamente como un solo objeto. En notación matemática, usualmente se escriben los puntos entre paréntesis, con una coma separando las coordenadas. Por ejemplo, $(0, 0)$ indica el origen, y (x, y) indica el punto x unidades a la derecha e y unidades hacia arriba del origen. En Java, un punto es representado por un objeto `Point`. Para crear un nuevo punto, tenés que usar el comando `new`:

```
Point nada;  
nada = new Point (3, 4);
```

La primera línea es una declaración de variable convencional: `nada` tiene tipo `Point`. La segunda línea tiene una apariencia un poco extraña; llama al comando `new`, especifica el tipo de objeto, y provee parámetros. Probablemente no te sorprenda que los parámetros sean las coordenadas del nuevo punto, $(3, 4)$.

El resultado del comando `new` es una **referencia** al nuevo punto. Explicaré más sobre referencias luego; por ahora lo importante es que la variable `nada` contiene una referencia al objeto recién creado. Hay una forma estándar de diagramar esta asignación, que es la que se muestra en la figura.



3. N.d.T: Kit de Herramientas de Ventana Abstracta.

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

La caja grande muestra el objeto recién creado con los dos valores dentro de él. Los nombres `x` e `y` son los nombres de las **variables de instancia**.

Tomados juntos, todas las variables, valores, y objetos en un programa se llaman **estado**. Diagramas como este que muestran el estado del programa se llaman **diagramas de estado**. A medida que el programa se ejecuta, el estado cambia, de modo que deberías pensar a un diagrama de estado como una fotografía de un punto particular en la ejecución.

8.4 Variables de instancia

A los fragmentos de datos que constituyen un objeto, a veces, se los llama componentes, registros o campos. En Java se los llama variables de instancia porque cada objeto, que es una **instancia** de su tipo, tiene su propia copia de las variables de instancia.

Es como la guantera de un auto. Cada auto es una instancia del tipo “auto”, y cada auto tiene su propia guantera. Si me pidieras sacar algo de la guantera de tu auto, me tendrías que decir primero, cuál auto es el tuyo.

De forma similar, si querés leer un valor de una variable de instancia, tenés que especificar el objeto del cual querés leerlo. En Java esto se logra con la “notación de punto”.

```
int x = nada.x;
```

La expresión `nada.x` significa “andá al objeto al que referencia `nada`, y obtené el valor de `x`.” En este caso, le asignamos ese valor a una variable local llamada `x`. Notá que no hay ningún conflicto entre la variable local llamada `x` y la variable de instancia llamada `x`. El propósito de la notación de punto es especificar a *qué* variable te referís sin ambigüedades.

Podés usar la notación de punto como parte de cualquier expresión de Java, de modo que las siguientes son válidas.

```
System.out.println (nada.x + ", " + nada.y);  
int distancia= nada.x * nada.x + nada.y * nada.y;
```

La primera línea imprime 3, 4; la segunda línea calcula el valor 25.

8.5 Objetos como parámetros

Podés pasar objetos como parámetros en la forma usual. Por ejemplo

```
public static void imprimirPunto (Point p) {
    System.out.println("(" + p.x + ", " + p.y + ")");
}
```

es un método que toma un punto como parámetro y lo imprime en el formato estándar. Si llamás a `printPoint (nada)`, imprimirá (3, 4). En realidad, Java tiene un método preincorporado para imprimir objetos de tipo `Point`. Si llamás a `System.out.println (nada)`, obtenés

```
java.awt.Point[x=3,y=4]
```

Este es un formato estándar que usa Java para imprimir objetos. Imprime el nombre del tipo, seguido por los contenidos del objeto, incluyendo los nombres y los valores de las variables de instancia.

Como segundo ejemplo, podemos reescribir el método `distancia` de la Sección 5.2 de modo que tome dos `Point` como parámetros en lugar de nuestros cuatro `double`.

```
public static double distancia(Point p1, Point p2) {
    double dx = (double)(p2.x - p1.x);
    double dy = (double)(p2.y - p1.y);
    return Math.sqrt (dx*dx + dy*dy);
}
```

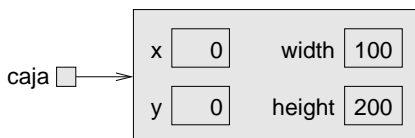
Los casteos de tipo no son necesarios; los agregué sólo como recordatorio de que las variables de instancia en `Point` son enteros.

8.6 Rectángulos

Los objetos `Rectangle` son similares a los puntos, excepto que tienen cuatro variables de instancia, llamadas `x`, `y`, `width`⁴ y `height`⁵. Además de eso, todo lo demás es prácticamente igual.

```
Rectangle caja = new Rectangle (0, 0, 100, 200);
```

crea un nuevo objeto `Rectangle` y hace que `caja` lo referencie. La figura muestra el efecto de la asignación.



4. N.d.T.: Ancho.

5. N.d.T.: Alto.

Si imprimís caja, obtenés

```
java.awt.Rectangle[x=0,y=0,width=100,height=200]
```

De nuevo, este es el resultado del método de Java que sabe cómo imprimir objetos Rectangle.

8.7 Objetos como tipo de retorno

Podés escribir métodos que devuelvan objetos. Por ejemplo, el método `buscarCentro` toma un `Rectangle` como parámetro y devuelve un `Point` que contiene las coordenadas del centro del rectángulo:

```
public static Point buscarCentro(Rectangle caja) {  
    int x = caja.x + caja.width/2;  
    int y = caja.y + caja.height/2;  
    return new Point (x, y);  
}
```

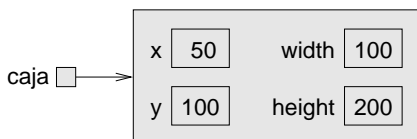
Notar que podés usar el `new` para crear un objeto, y luego inmediatamente usar el resultado como valor de retorno.

8.8 Los objetos son mutables

Podés cambiar el contenido de un objeto haciendo una asignación a una de sus variables de instancia. Por ejemplo, para “mover” un rectángulo sin cambiar su tamaño, podés modificar los valores de `x` e `y`:

```
caja.x = caja.x + 50;  
caja.y = caja.y + 100;
```

El resultado se muestra en la figura:



Podemos tomar este código y encapsularlo en un método, y generalizarlo para mover un rectángulo cualquier distancia:

```
public static void moverRect(Rectangle caja, int dx, int dy) {  
    caja.x = caja.x + dx;  
    caja.y = caja.y + dy;  
}
```

Las variables `dx` y `dy` indican cuánto mover el rectángulo en cada dirección. Llamar a este método tiene el efecto de modificar el `Rectangle` que se pasa como parámetro.

```
Rectangle caja = new Rectangle (0, 0, 100, 200);  
moverRect (caja, 50, 100);  
System.out.println (caja);
```

imprime `java.awt.Rectangle[x=50,y=100,width=100,height=200]`.

Modificar objetos pasándolos como parámetros a métodos puede ser útil, pero también puede hacer a la depuración más difícil porque no siempre está claro qué método modifica sus argumentos y cuál no lo hace. Luego, discutiré algunos pros y contras de este estilo de programación.

Mientras tanto, podemos disfrutar del lujo de los métodos preincorporados de Java, que incluyen `translate`, que hace exactamente lo mismo que `moverRect`, a pesar de que la sintaxis para llamarlo es ligeramente distinta. En lugar de pasar el `Rectangle` como parámetro, llamamos a `translate` en el `Rectangle` y pasamos sólo `dx` y `dy` como parámetros.

```
caja.translate (50, 100);
```

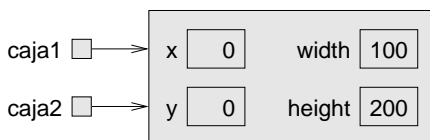
El efecto es exactamente el mismo.

8.9 Aliasing

Recordá que cuando hacés una asignación a una variable del tipo de algún objeto, estás asignando una *referencia* al objeto. Es posible tener múltiples variables que referencian al mismo objeto. Por ejemplo, este código:

```
Rectangle caja1 = new Rectangle (0, 0, 100, 200);  
Rectangle caja2 = caja1;
```

genera un diagrama de estado como este:



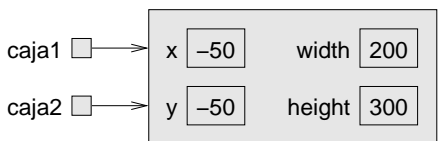
Tanto `caja1` como `caja2` referencian o “apuntan” al mismo objeto. En otras palabras, este objeto tiene dos nombres, `caja1` y `caja2`. Cuando una

persona tiene dos nombres, se dice que tienen un **alias**. Lo mismo ocurre con los objetos, que se dice que hacen **aliasing**.

Cuando dos variables apuntan al mismo objeto, cualquier cambio que afecta una variable, también afecta a la otra. Por ejemplo:

```
System.out.println (caja2.width);  
caja1.grow (50, 50);  
System.out.println (caja2.width);
```

La primera línea imprime 100, que es el ancho del `Rectangle` referenciado por `caja2`. La segunda línea llama al método `grow` en `caja1`, expande el rectángulo en 50 píxeles en cada dirección (ver la documentación para más detalles). El efecto se muestra en la figura:



Como debería quedar claro de esta figura, cualquier cambio que se haga sobre `caja1` también se aplica sobre `caja2`. Luego, el valor impreso en la tercera línea es 200, el ancho del rectángulo expandido. (Como comentario aparte, es perfectamente válido que las coordenadas de un `Rectangle` sean negativas.)

Como podés deducir de este ejemplo simple, el código que involucra `aliasing` puede volverse confuso con rapidez, y puede ser muy difícil de depurar. En general, el `aliasing` debe ser evitado o usado con precaución.

8.10 null

Cuando creás una variable del tipo objeto, recordá que estás creando una *referencia* a un objeto. Hasta que hagas que esa variable apunte a un objeto, el valor de esa variable es `null`. `null` es un valor especial en Java (y una palabra reservada) que es utilizada para simbolizar “ningún objeto.”

La declaración `Point nada;` es equivalente a esta inicialización:

```
Point nada = null;
```

y se muestra en el siguiente diagrama de estado:



El valor `null` está representado por un punto sin ninguna flecha.

Si tratás de usar un objeto nulo (que apunta a null), tanto accediendo a una variable de instancia como llamando a un método, obtendrás una `NullPointerException`. El sistema imprimirá un mensaje de error y terminará el programa.

```
Point nada = null;
int x = nada.x;           // NullPointerException
nada.translate (50, 50);  // NullPointerException
```

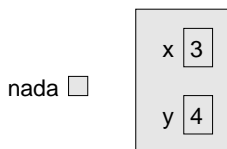
Por otro lado, es válido pasar un objeto nulo como parámetro, o recibirlo como resultado de un método. De hecho, es común hacerlo, por ejemplo, para representar un conjunto vacío o una condición de error.

8.11 Recolector de basura

En la Sección 8.9 hablamos de qué ocurre cuando más de una variable referencia a un mismo objeto. ¿Qué ocurre cuando *ninguna* variable referencia a un objeto? Por ejemplo:

```
Point nada = new Point (3, 4);
nada = null;
```

La primera línea crea un objeto `Point` y hace que `nada` apunte a él. La segunda línea cambia `nada` de modo que, en lugar de referenciar al objeto, no referencia a nada (o bien, al objeto nulo)



Si nadie referencia a un objeto, entonces nadie puede escribir o leer sus valores, o llamar a un método en él. De hecho, deja de existir. Podríamos mantener el objeto en memoria, pero sólo desperdiciaría espacio, con lo que periódicamente, a medida que tu programa se ejecuta, el sistema de Java busca objetos perdidos y los libera, en un proceso llamado **recolección de basura**⁶. Luego, la memoria utilizada por el objeto volverá a estar disponible para ser utilizada como parte de un nuevo objeto.

No necesitás hacer nada para que funcione la recolección de basura y, en general, no serás consciente de ella.

6. N.d.T.: *garbage collection* en inglés.

8.12 Objetos y tipos primitivos

Hay dos clases de tipos en Java, los primitivos y los tipos de objetos. Los primitivos, como `int` y `boolean` comienzan con minúscula; los tipos de objetos comienzan con mayúscula. Esta distinción es útil porque nos recuerda algunas de las diferencias entre ellos:

- Cuando declararás una variable de un tipo primitivo, obtendrás espacio de almacenamiento para un valor primitivo. Cuando declararás una variable de objeto, obtendrás espacio para referenciar a un objeto. Para obtener el espacio para el objeto propiamente dicho, necesitarás usar el comando `new`.
- Si no inicializas una variable de un tipo primitivo, se le da un valor por defecto que depende del tipo. Por ejemplo, `0` para los `int` y `false` para los `boolean`. El valor por defecto para las variables de tipo objeto es `null`, lo cual indica “ningún objeto”.
- Las variables primitivas están bien aisladas en el sentido de que no hay nada que puedas hacer en un método que pueda afectar una variable en otro método. Las variables de objeto pueden ser más complicadas para trabajar porque no están bien aisladas. Si pasas una referencia a un objeto como parámetro, el método que llamas puede modificar el objeto, en cuyo caso verás el efecto que tuvo la llamada al método sobre el objeto referenciado. Lo mismo ocurre cuando llamas a un método de un objeto. Desde ya que esto puede ser algo bueno, pero debes estar al tanto de este comportamiento.

Hay otra diferencia entre los tipos primitivos y de objeto. No puedes agregar nuevos tipos primitivos al lenguaje Java (a menos que consigas entrar en el comité de estandarización), pero ¡puedes crear nuevos tipos de objetos! Veremos cómo en el próximo capítulo.

8.13 Glosario

paquete: Colección de clases. Las clases preincorporadas de Java están organizadas en paquetes.

AWT: Siglas en inglés para “Abstract Window Toolkit”, uno de los más grandes y más comúnmente usados paquetes de Java.

instancia: Ejemplo de una categoría. Mi gato es una instancia de la categoría “cosas felinas”. Cada objeto es una instancia de una clase.

variable de instancia: Variable que forma parte de un objeto. Cada objeto (instancia) tiene su propia copia de las variables de instancia para su respectiva clase.

referencia: Valor que indica un objeto. En un diagrama de estado, una referencia aparece como una flecha.

aliasing: Condición en la cual dos o más variables referencian un mismo objeto.

recolección de basura: Proceso de encontrar objetos que no se referencian y recuperar el espacio de memoria en el que se almacenaban.

estado: Descripción completa de todas las variables y objetos y sus valores, en un punto dado de la ejecución de un programa.

diagrama de estado: Fotografía del estado de un programa, mostrada gráficamente.

8.14 Ejercicios

Ejercicio 8.1

- Para el siguiente programa, dibujá un diagrama de estado mostrando las variables locales, y los parámetros de main y pepe, y mostrá qué objetos referencian esas variables.
- ¿Cuál es la salida del programa?

```
public static void main (String[] args)
{
    int x = 5;
    Point nada = new Point (1, 2);

    System.out.println (pepe (x, nada));
    System.out.println (x);
    System.out.println (nada.x);
    System.out.println (nada.y);
}

public static int pepe (int x, Point p)
{
    x = x + 7;
    return x + p.x + p.y;
}
```

El objetivo del ejercicio es asegurarte que entiendas el mecanismo para pasar objetos como parámetros.

Ejercicio 8.2

- a. Para el siguiente programa, dibuja un diagrama de estado mostrando el estado del programa justo antes de que distancia termine. Incluye todas las variables y parámetros y objetos a los cuales referencien esas variables.
- b. ¿Cuál es la salida de este programa?

```
public static double distancia (Point p1, Point p2) {
    int dx = p1.x - p2.x;
    int dy = p1.y - p2.y;
    return Math.sqrt (dx*dx + dy*dy);
}

public static Point buscarCentro (Rectangle caja) {
    int x = caja.x + caja.width/2;
    int y = caja.y + caja.height/2;
    return new Point (x, y);
}

public static void main (String[] args) {
    Point nada = new Point (5, 8);

    Rectangle rect = new Rectangle (0, 2, 4, 4);
    Point centro = buscarCentro(rect);

    double dist = distancia (centro, nada);

    System.out.println (dist);
}
```

Ejercicio 8.3

El método `grow` forma parte de la clase preincorporada `Rectangle`. He aquí la documentación del mismo (tomada de la página de Sun) traducida al español:

```
public void grow(int h, int v)
```

Aumenta el tamaño del rectángulo tanto vertical como horizontalmente.

Este método modifica el rectángulo de modo que es `h` unidades más ancho tanto el lado izquierdo como

el derecho, y v unidades más alto tanto arriba como abajo.

El nuevo rectángulo tiene su esquina superior izquierda en $(x - h, y - v)$, un ancho igual al ancho + $2h$, y un alto igual al alto + $2v$.

Si se pasan valores negativos para h y v , el tamaño del rectángulo decrece en forma acorde a lo anterior. El método `grow` no verifica que los valores resultantes de ancho y alto no sean negativos.

- a. ¿Cuál es la salida del siguiente programa?
- b. Dibujá un diagrama de estado que muestre el estado del programa justo antes de que termine `main`. Incluí todas las variables locales y los objetos que referencian.
- c. Al final de `main`, `p1` y `p2` ¿hacen aliasing? ¿Por qué sí o por qué no?

```
public static void imprimirPunto (Point p) {
    System.out.println("(" + p.x + ", " + p.y + ")");
}

public static Point buscarCentro (Rectangle caja) {
    int x = caja.x + caja.width/2;
    int y = caja.y + caja.height/2;
    return new Point (x, y);
}

public static void main (String[] args) {

    Rectangle caja1 = new Rectangle (2, 4, 7, 9);
    Point p1 = buscarCentro (caja1);
    imprimirPunto (p1);

    caja1.grow (1, 1);
    Point p2 = buscarCentro (caja1);
    imprimirPunto (p2);
}
```

Ejercicio 8.4

Probablemente a esta altura, ya te estás aburriendo del método factorial, pero vamos a hacer una versión más.

- a. Creá un programa nuevo llamado `Big.java` y comenzá escribiendo una versión iterativa de factorial.

- b. Imprimí una tabla de enteros de 0 a 30 junto con sus factoriales. En un punto determinado, alrededor del 15, probablemente notarás que las respuestas ya no son correctas. ¿Por qué?
- c. `BigInteger` es un tipo de objeto preincorporado que puede representar enteros arbitrariamente grandes. No hay cota superior excepto por las limitaciones de memoria y velocidad de procesamiento. Leé la documentación para la clase `BigInteger` en el paquete `java.math`.
- d. Hay varias formas de crear un nuevo `BigInteger`, pero la que recomiendo usa `valueOf`. El siguiente código convierte un `int` en un `BigInteger`:

```
int x = 17;
BigInteger grande = BigInteger.valueOf (x);
```

Escribí este código y probá algunos casos simples como crear un `BigInteger` e imprimirlo. ¡Notá que `println` sabe cómo imprimir `BigIntegers`! No te olvides de añadir `import java.math.BigInteger` al inicio del programa.

- e. Desafortunadamente, dado que `BigInteger` no es un tipo primitivo, no podemos usar los operadores matemáticos en ellos. En cambio, debemos utilizar métodos como, por ejemplo, `add`. Para poder sumar dos `BigIntegers`, tenés que llamar a `add` sobre uno de ellos y pasar el otro como argumento. Por ejemplo:

```
BigInteger chico = BigInteger.valueOf (17);
BigInteger grande = BigInteger.valueOf (1700000000);
BigInteger total = chico.add (grande);
```

Probá alguno de los otros métodos, como `multiply`⁷ y `pow`⁸.

- f. Modificar factorial de modo que efectúe su cálculo usando `BigIntegers`, y luego devuelve el `BigInteger` como resultado. El parámetro puede quedar sin modificar — seguirá siendo un `int`.
- g. Intentá imprimir la tabla nuevamente con tu versión modificada de la función factorial. ¿Es correcta hasta el 30? ¿Hasta dónde podés llegar? Yo calculé el factorial de todos los números de 0 a 999, pero mi máquina es bastante lenta, de modo que tomó un tiempo. El último número, 999!, tiene 2565 dígitos.

Ejercicio 8.5

Muchos algoritmos de encriptación dependen de la habilidad de elevar enteros muy grandes a una potencia entera. Aquí hay un método que implementa un método (razonablemente) rápido para exponenciación entera:

7. N.d.T.: mutiplicar.

8. N.d.T.: potencia.

```

public static int potencia (int x, int n) {
    if (n==0) return 1;

    // calcular x a la n/2 recursivamente
    int t = potencia (x, n/2);

    // si n es par, el resultado es t al cuadrado
    // si n es impar, el resultado es t al cuadrado por x

    if (n%2 == 0) {
        return t*t;
    } else {
        return t*t*x;
    }
}

```

El problema con este método es que sólo funciona si el resultado es menor a dos mil millones. Reescribirlo de modo que el resultado sea un `BigInteger`. Los parámetros deben continuar siendo `int`.

Podés usar los métodos `add` y `multiply` de `BigInteger`, pero no uses el método `pow`, porque arruinaría la diversión.

Capítulo 9

Creá tus propios objetos

9.1 Definiciones de clases y tipos de objetos

Cada vez que escribís una definición de una clase, creás un nuevo tipo de objeto, cuyo nombre es el mismo que el de la clase. Mucho antes, en la Sección 1.5, cuando definimos la clase llamada `Hola`, también creamos un tipo de objeto llamado `Hola`. No creamos ninguna variable con el tipo `Hola`, y no usamos el comando `new` para crear un objeto `Hola`, pero ¡podríamos haberlo hecho!

Ese ejemplo no tiene mucho sentido, ya que no hay razón para crear un objeto `Hola`, y no está claro para qué serviría si lo hiciéramos. En este capítulo, veremos algunos ejemplos de definiciones de clases que crean nuevos tipos de objeto *útiles*.

Estas son las ideas más importantes de este capítulo:

- Definir una nueva clase crea también un nuevo tipo de objeto con el mismo nombre.
- Una definición de una clase es como una plantilla para objetos: determina qué variables de instancia tienen y qué métodos pueden operar en ellos.
- Todo objeto pertenece a un tipo de objeto; con lo cual, es una instancia de alguna clase.
- Cuando llamás al comando `new` para crear un objeto, Java llama a un método especial llamado **constructor** para darle un valor inicial a las variables de instancia. Podés proporcionar uno o más constructores en la definición de la clase.

- Típicamente, todos los métodos que operan sobre un tipo van en la definición de la clase de ese tipo.

Estos son algunos inconvenientes sintácticos de la definición de clases:

- Los nombres de clases (y por lo tanto, los tipos de los objetos) siempre comienzan con una mayúscula, lo que ayuda a distinguirlos de tipos primitivos y nombres de variables.
- Usualmente se pone una definición de clase por archivo, y el nombre del archivo debe ser el mismo que el nombre de la clase, con el sufijo `.java`. Por ejemplo, la clase `Tiempo` está definida en un archivo llamado `Tiempo.java`.
- En cualquier programa, una clase se designa como la **clase de inicio**. La clase de inicio debe contener un método llamado `main`, en el cual comienza la ejecución del programa. Otras clases *pueden* tener un método llamado `main`, pero no será ejecutado.

Con esos inconvenientes aclarados, miremos un ejemplo de un tipo definido por el usuario, `Tiempo`.

9.2 Tiempo

Una motivación común para crear un nuevo tipo de objeto es la de tomar varios datos relacionados y encapsularlos en un objeto que puede ser manipulado (pasado como argumento, operado) como una unidad. Ya hemos visto dos tipos preincorporados así: `Point` y `Rectangle`.

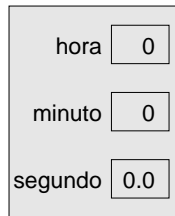
Otro ejemplo, que implementaremos nosotros, es `Tiempo`, que se utiliza para grabar la hora del día. Los datos que forman el tiempo son la hora, el minuto y el segundo. Dado que todo objeto `Tiempo` contendrá estos datos, necesitamos crear variables de instancia para almacenarlos.

El primer paso es decidir qué tipo debería tener cada variable. Parece claro que hora y minuto deberían ser enteros. Sólo para hacer las cosas más interesantes, hagamos que segundo sea `double`, de modo que podamos guardar fracciones de segundo.

Las variables de instancia se declaran al principio de la definición de la clase, fuera de toda definición de método:

```
class Tiempo {
    int hora, minuto;
    double segundo;
}
```

Por sí mismo, este fragmento de código es una definición válida para una clase. El diagrama de estados para un objeto Tiempo se vería así:



Luego de declarar las variables de instancia, el siguiente paso usualmente consiste en definir un constructor para la nueva clase.

9.3 Constructores

El rol usual del constructor es inicializar las variables de instancia. La sintaxis para los constructores es similar a la otros métodos, con tres excepciones:

- El constructor lleva el mismo nombre que la clase.
- Los constructores no tienen tipo de retorno ni devuelven nada.
- La palabra reservada `static` no va.

Aquí hay un ejemplo para la clase `Tiempo`:

```
public Tiempo () {
    this.hora = 0;
    this.minuto = 0;
    this.segundo = 0.0;
}
```

Notá que donde esperarías ver el tipo de retorno, entre `public` y `Tiempo`, no hay nada. Esa es la manera en que nosotros (y el compilador) podemos identificar a este método como un constructor.

Este constructor no toma ningún parámetro, como lo indican los paréntesis vacíos `()`. Cada línea del constructor inicializa una variable de instancia a un valor arbitrario (en este caso, medianoche). El nombre `this` es una palabra reservada especial con la cual podemos hablar del objeto que estamos creando. Podés usar `this` de la misma manera que usás el nombre de cualquier otro objeto. Por ejemplo, podés leer y escribir variables de instancia de `this`, y podés pasar `this` como parámetro a otros métodos.

Pero `this` no se declara y no se usa `new` para crearlo. De hecho, ¡ni siquiera podés asignarle algo! A `this` lo crea el sistema; todo lo que tenés que hacer es guardar valores en sus variables de instancia.

Un error común cuando se escriben constructores es poner una sentencia `return` al final. Resistí la tentación.

9.4 Más constructores

Los constructores pueden ser sobrecargados, como cualquier otro método, lo que significa que podés proporcionar múltiples constructores con diferentes parámetros. Java sabe qué constructor llamar al buscar cuál de ellos se corresponde con los parámetros que se le pasaron al comando `new`.

Es muy común tener un constructor que no toma argumentos (como el anterior), y uno que toma una lista de parámetros que es idéntica a la lista de variables de instancia. Por ejemplo:

```
public Tiempo(int hora, int minuto, double segundo) {
    this.hora= hora;
    this.minuto = minuto;
    this.segundo = segundo;
}
```

Los nombres y los tipos de los parámetros son exactamente los mismos que los nombres y los tipos de las variables de instancia. Todo lo que hace el constructor es copiar la información de los parámetros a las variables de instancia.

Si vas atrás y mirás la documentación de `Point` y `Rectangle`, verás que ambas clases proveen constructores como este. Sobrecargar los constructores da la flexibilidad para crear un objeto primero y llenar los espacios vacíos luego, u obtener toda la información antes de crear el objeto.

Hasta ahora esto puede no parecer muy interesante, y de hecho, no lo es. Escribir constructores es un proceso aburrido y mecánico. Una vez que hayas escrito un par, vas a ver que vas a poder escribirlos hasta durmiendo y que sólo necesitás mirar la lista de variables de instancia.

9.5 Creando un objeto nuevo

A pesar de que los constructores se vean como métodos, nunca los llamas directamente. Sino que, cuando usás el comando `new`, el sistema reserva espacio para el nuevo objeto y luego llama a tu constructor para inicializar las variables de instancia.

El siguiente programa muestra dos maneras para crear e inicializar objetos `Tiempo`:

```

class Tiempo {
    int hora, minuto;
    double segundo;

    public Tiempo () {
        this.hora = 0;
        this.minuto = 0;
        this.segundo = 0.0;
    }

    public Tiempo (int hora, int minuto, double segundo) {
        this.hora = hora;
        this.minuto = minuto;
        this.segundo = segundo;
    }

    public static void main (String[] args) {

        // una forma de crear e inicializar un objeto Tiempo
        Tiempo t1 = new Tiempo ();
        t1.hora = 11;
        t1.minuto = 8;
        t1.segundo = 3.14159;
        System.out.println (t1);

        // otra manera de hacer lo mismo
        Tiempo t2 = new Tiempo (11, 8, 3.14159);
        System.out.println (t2);
    }
}

```

Como ejercicio, deducí el flujo de ejecución de este programa.

En main, la primera vez que llamamos al comando new, no pasamos ningún parámetro, con lo cual Java llama al primer constructor. Las líneas inmediatamente a continuación le asignan valores a cada una de las variables de instancia.

La segunda vez que llamamos al comando new, le pasamos parámetros que se corresponden con los parámetros del segundo constructor. Esta manera de inicializar las variables de instancia es más concisa (y ligeramente más eficiente), pero puede ser más difícil de leer, dado que no está claro qué valores se le asignan a qué variables de instancia.

9.6 Imprimiendo un objeto

La salida del programa anterior es:

```
Tiempo@80cc7c0
```

```
Tiempo@80cc807
```

Cuando Java imprime el valor de un objeto de un tipo definido por el usuario, imprime el nombre del tipo y un código especial hexadecimal (base 16) que es único para cada objeto. Este código no es significativo por sí mismo; de hecho, puede variar de máquina a máquina y de ejecución a ejecución. Pero puede ser útil para la depuración, para mantener el rastro de objetos particulares.

Para imprimir objetos de una manera que sea más significativa para los usuarios (al contrario que para los programadores), usualmente querés escribir un método llamado algo así como `imprimirTiempo`:

```
public static void imprimirTiempo (Tiempo t) {  
    System.out.println (t.hora + ":" + t.minuto  
                        + ":" + t.segundo);  
}
```

Compará este método con el `imprimirTiempo` de la Sección 3.10.

La salida de este método, si le pasamos tanto `t1` como `t2` como parámetro, es `11:8:3.14159`. A pesar de que esto es reconocible como una hora, no está en un formato muy estándar. Por ejemplo, si la cantidad de minutos o segundos es menor a 10, esperamos un 0 a la izquierda para mantener el largo. Además, quisiéramos tirar la parte decimal de los segundos. En otras palabras, queremos algo como `11:08:03`.

En muchos lenguajes, hay formas simples para controlar el formato de salida para los números. En Java no hay formas simples.

Java provee herramientas muy potentes para imprimir cosas con formato como fechas y horas, y también para interpretar valores formateados. Desafortunadamente, estas herramientas no son muy fáciles de usar, con lo que voy a dejarlas fuera de este libro. Si querés, podés mirar la documentación de la clase `Date` en el paquete `java.util`.

9.7 Operaciones sobre objetos

A pesar de que no podemos imprimir horas en un formato óptimo, aún podemos escribir métodos para manipular objetos `Tiempo`. En las próximas secciones, mostraré distintas formas que pueden adquirir métodos que operan sobre objetos. Para algunas operaciones, tendrás que elegir entre varias alternativas, con lo que deberías considerar los pros y los contras de:

función pura: Toma un objeto y/o un valor de un tipo primitivo como parámetro pero no modifica los objetos. El valor que devuelve es o bien un valor primitivo o bien un nuevo objeto creado dentro del método.

modificador: Toma objetos como parámetros y modifica algunos o todos ellos. Frecuentemente devuelve void.

método de llenado: Uno de sus argumentos es un objeto “vacío” que este método se encarga de llenar. Técnicamente, es un tipo de modificador.

9.8 Funciones puras

Un método es considerado una función pura si el resultado depende sólo de sus argumentos, y no tiene efectos secundarios como modificar un parámetro o imprimir algo. El único resultado de llamar a una función pura es el valor que devuelve.

Un ejemplo es posterior, que compara dos Tiempos y devuelve un boolean que indica si el primer operando es posterior al segundo:

```
public static boolean posterior (Tiempo tiempo1, Tiempo tiempo2) {
    if (tiempo1.hora > tiempo2.hora) return true;
    if (tiempo1.hora < tiempo2.hora) return false;

    if (tiempo1.minuto > tiempo2.minuto) return true;
    if (tiempo1.minuto < tiempo2.minuto) return false;

    if (tiempo1.segundo > tiempo2.segundo) return true;
    return false;
}
```

¿Cuál es el resultado de este método si los dos tiempos son iguales? ¿Te parece un resultado apropiado para este método? Si estuvieras escribiendo la documentación, ¿mencionarías ese caso específicamente?

Un segundo ejemplo es sumarTiempo, que calcula la suma de dos tiempos. Por ejemplo, si son las 9:14:30, y tu lavarropas tarda 3 horas y 35 minutos en terminar, podés usar sumarTiempo para calcular cuándo va a finalizar el lavado. Este sería un borrador del método que no es totalmente correcto:

```
public static Tiempo sumarTiempo(Tiempo t1, Tiempo t2) {
    Tiempo sum = new Tiempo ();
    sum.hora = t1.hora + t2.hora;
```

```

        sum.minuto = t1.minuto + t2.minuto;
        sum.segundo = t1.segundo + t2.segundo;
        return sum;
    }

```

A pesar de que este método devuelve un objeto `Tiempo`, no es un constructor. Deberías volver y comparar la sintaxis de un método como este con la sintaxis de un constructor, porque es fácil confundirse.

Aquí hay un ejemplo de cómo usar este método. Si `tiempoActual` contiene la hora actual y `tiempoLavadora` contiene el tiempo que le toma a tu lavadora en terminar, entonces podés usar `sumarTiempo` para calcular la hora de finalización:

```

Time tiempoActual = new Time (9, 14, 30.0);
Time tiempoLavadora = new Time (3, 35, 0.0);
Time tiempoFinalizacion = addTime (tiempoActual, tiempoLavadora);
imprimirTiempo (tiempoFinalizacion);

```

La salida de este programa es 12:49:30.0, que es correcta. Por otro lado, hay casos en los cuales el resultado no es correcto. ¿Se te ocurre uno?

El problema es que este método no contempla casos en los que el número de segundos o minutos suma 60 o más. En ese caso, tenemos que “llevar” los segundos extra en la columna de minutos, o los minutos extra en la columna de horas. He aquí una segunda versión corregida de este método.

```

public static Time sumarTiempo (Tiempo t1, Tiempo t2) {
    Tiempo sum = new Tiempo ();
    sum.hora = t1.hora + t2.hora;
    sum.minuto = t1.minuto + t2.minuto;
    sum.segundo = t1.segundo + t2.segundo;

    if (sum.segundo >= 60.0) {
        sum.segundo -= 60.0;
        sum.minuto += 1;
    }
    if (sum.minuto >= 60) {
        sum.minuto -= 60;
        sum.hora += 1;
    }
    return sum;
}

```

A pesar de que es correcto, está comenzando a alargarse un poco. Más tarde sugeriré un enfoque alternativo a este problema que resultará mucho más corto.

Este código muestra dos operadores que no habíamos visto antes, `+=` y `-=`. Estos operadores proveen una forma de incrementar y decrementar variables. Son muy parecidos a `++` y `--`, excepto (1) funcionan también con `double` además de `int`, y (2) la cantidad del incremento no tiene que ser necesariamente 1. La sentencia `sum.segundo -= 60.0;` es equivalente a `sum.segundo = sum.segundo - 60;`.

9.9 Modificadores

Como ejemplo de un modificador, considerará el método `incrementar`, que añade una cantidad dada de segundos a un objeto `Tiempo`. Nuevamente, un boceto aproximado de este método sería:

```
public static void incrementar (Tiempo tiempo, double segs) {
    tiempo.segundo += segs;

    if (tiempo.segundo >= 60.0) {
        tiempo.segundo -= 60.0;
        tiempo.minuto += 1;
    }
    if (tiempo.minuto >= 60) {
        tiempo.minuto -= 60;
        tiempo.hora += 1;
    }
}
```

La primera línea efectúa la operación básica; el resto se encarga de los mismos casos que vimos antes.

¿Es correcto este método? ¿Qué ocurre cuando el parámetro `segs` es bastante más grande que 60? En ese caso, no es suficiente con restar 60 una única vez; tenemos que seguir haciéndolo hasta que `segundo` sea menor a 60. Podemos hacerlo simplemente reemplazando las sentencias `if` por sentencias `while`:

```
public static void incrementar (Tiempo tiempo, double segs) {
    tiempo.segundo += segs;

    while (tiempo.segundo >= 60.0) {
        tiempo.segundo -= 60.0;
        tiempo.minuto += 1;
    }
}
```



```

while (tiempo.minuto >= 60) {
    tiempo.minuto -= 60;
    tiempo.hora += 1;
}
}

```

Esta solución es correcta, pero no muy eficiente. ¿Se te ocurre una solución que no requiera iteración?

9.10 Métodos de llenado

Ocasionalmente verás métodos como `sumarTiempo` escritos con una interfaz distinta (distintos argumentos y tipo de retorno). En lugar de crear un nuevo objeto cada vez que se llama a `sumarTiempo`, podemos pedir al que llama al método que provea un objeto “vacío” donde `sumarTiempo` puede almacenar el resultado. Comparar esta versión con la anterior:

```

public static void sumarTiempoLlenado (Tiempo t1, Tiempo t2,
                                         Tiempo sum) {

    sum.hora = t1.hora + t2.hora;
    sum.minuto = t1.minuto + t2.minuto;
    sum.segundo = t1.segundo + t2.segundo;

    if (sum.segundo >= 60.0) {
        sum.segundo -= 60.0;
        sum.minuto += 1;
    }
    if (sum.minuto >= 60) {
        sum.minuto -= 60;
        sum.hora += 1;
    }
}

```

Una ventaja de este enfoque es que quien llama al método tiene la opción de reutilizar el mismo objeto varias veces para efectuar varias sumas en serio. Esto puede ser ligeramente más eficiente, aunque puede ser suficientemente confuso como para causar errores sutiles. Para la vasta mayoría de la programación, vale la pena sacrificar un poco de tiempo de ejecución para evitar perder mucho más tiempo en depuración.

9.11 ¿Cuál es mejor?

Cualquier cosa que pueda hacerse con modificadores y métodos de llenado puede hacerse también con funciones puras. De hecho, hay lenguajes de programación, llamados lenguajes **funcionales**, que sólo permiten funciones puras. Algunos programadores creen que los programas

que usan sólo funciones puras son más fáciles de desarrollar y menos propensos a errores que aquellos que usan modificadores. No obstante, hay veces en los cuales los modificadores son convenientes, y casos en los que los programas funcionales son menos eficientes.

En general, te recomiendo escribir funciones puras siempre que sea razonable hacerlo, y recurrir a los modificadores sólo si hay una ventaja notable en ello. Este enfoque puede llamarse un estilo de programación funcional.

9.12 Desarrollo incremental vs. planificación

En este capítulo he mostrado un enfoque para el desarrollo de un programa al que denomino **prototipado rápido con mejora iterativa**. En cada caso, escribí un boceto aproximado (o prototipo) que efectuaba la operación básica, y luego probándolo en algunos casos, corrigiendo las fallas a medida que las encontraba.

A pesar de que este enfoque puede ser muy efectivo, también puede llevar a código innecesariamente complicado—pues contempla muchos casos particulares—y poco confiable—pues es difícil convencerte de que encuentras *todos* los errores.

Una alternativa es la planificación de alto nivel, en la cual una pequeña comprensión del problema puede hacer mucho más fácil la programación. En este caso lo que debemos darnos cuenta es que un Tiempo es en realidad un número de tres dígitos en base 60. El segundo es la “columna de 1”, el minuto es la “columna de 60”, y la hora es la “columna de 3600”.

Cuando escribimos sumarTiempo e incrementar, efectivamente estábamos haciendo una suma en base 60, razón por la cual tuvimos que “llevar” algo de una columna a la otra.

Por lo tanto, un enfoque alternativo para el problema es convertir los Tiempos en doubles y aprovechar que la computadora ya sabe hacer cálculos aritméticos con doubles. Aquí está el método que convierte un Tiempo en un double:

```
public static double convertirASegundos (Tiempo t) {  
    int minutos = t.hora * 60 + t.minuto;  
    double segundos = minutos * 60 + t.segundo;  
    return segundos;  
}
```

Ahora todo lo que necesitamos es una forma de convertir un double en un objeto Tiempo. Podríamos escribir un método para hacerlo, pero tendría mucho más sentido escribir un tercer constructor:

```

public Tiempo (double segs) {
    this.hora = (int) (segs / 3600.0);
    segs -= this.hora * 3600.0;
    this.minuto = (int) (segs / 60.0);
    segs -= this.minuto * 60;
    this.segundo = segs;
}

```

Este constructor es un poco diferente de los anteriores, dado que involucra algún cálculo junto con las asignaciones a las variables de instancia.

Quizás tengas que pensarlo un poco para convencerte que la técnica que estoy usando de una base a la otra es correcta. Asumiendo que estás convencido, podemos usar estos métodos para reescribir `sumarTiempo`:

```

public static Tiempo sumarTiempo (Tiempo t1, Tiempo t2) {
    double segundos = convertirASegundos(t1) +
                      convertirASegundos(t2);
    return new Tiempo (segundos);
}

```

Esto es mucho más corto que su versión original, y mucho más fácil de demostrar que es correcto (asumiendo, como siempre, que los métodos a los que llama son correctos). Como ejercicio, reescribí incrementar de manera análoga.

9.13 Generalización

En algún sentido, convertir de base 60 a base 10 y viceversa es más difícil que tan sólo tratar con tiempos. La conversión de base es más abstracta; nuestra intuición para pensar tiempos es mejor.

Pero si tenemos la comprensión para tratar tiempos como números en base 60, y hacemos la inversión de escribir los métodos de conversión (`convertirASegundos` y el tercer constructor), obtenemos un programa que es más corto, más fácil de leer y depurar, y más confiable.

También es más fácil añadir nuevas características. Por ejemplo, imaginá abstraer dos Tiempos para encontrar la duración entre ellos. El enfoque ingenuo sería implementar la substracción completa “pidiendo prestado” cuando sea el caso. Usar los métodos de conversión sería mucho más fácil.

Irónicamente, a veces hacer un problema más difícil (más general), lo hace más fácil (menos casos especiales, menos posibilidades de error).

9.14 Algoritmos

Cuando escribís una solución general para una clase de problemas, al contrario de una solución específica para un problema particular, escribís un **algoritmo**. Mencioné esta palabra en el Capítulo 1, pero no la definí con cuidado. No es fácil de definir, con lo que probaré desde varios ángulos.

Primero, considerá algunas cosas que no son algoritmos. Por ejemplo, cuando aprendiste a multiplicar números de un sólo dígito, probablemente memorizaste la tabla de multiplicación. De hecho, memorizaste 100 soluciones específicas, con lo que ese conocimiento no es realmente algorítmico.

Pero si eras “perezoso”, probablemente hiciste trampa aprendiendo algunos trucos. Por ejemplo, para encontrar el producto de n por 9, podés escribir $n - 1$ como el primer dígito y $10 - n$ como el segundo. Este truco es una solución general para multiplicar cualquier número de una sola cifra por 9. ¡Eso es un algoritmo!

De manera similar, las técnicas que aprendiste para suma con acarreo, resta “pidiendo prestado”, y división para varias cifras son todos algoritmos. Una de las características de los algoritmos es que no requieren ninguna inteligencia para llevarse a cabo. Son procesos mecánicos en los cuales cada paso continúa al anterior de acuerdo con un conjunto simple de reglas.

En mi opinión, es embarazoso que los humanos pasen tanto tiempo en la escuela aprendiendo a ejecutar algoritmos que, casi literalmente, no requieren inteligencia.

Por otro lado, el proceso de diseñar algoritmos es interesante, intelectualmente desafiante, y una parte central de lo que denominamos programación.

Muchas de las cosas que la gente hace naturalmente, sin dificultad o 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 la forma de un algoritmo.

Más adelante tendrás la oportunidad de diseñar algoritmos sencillos para una variedad de problemas.

9.15 Glosario

clase: Previamente, definí una clase como una colección de métodos relacionados. En este capítulo aprendimos que una definición de clase es también una plantilla para un nuevo tipo de objeto.

instancia: Miembro de una clase. Todo objeto es una instancia de alguna clase.

constructor: Método especial que inicializa las variables de instancia de un objeto recién construido.

proyecto: Colección de una o más definiciones de clase (una por archivo) que constituyen un programa.

clase de inicio: Clase que contiene el método `main` donde se inicia la ejecución del programa.

función pura: Método cuyo resultado depende sólo de sus parámetros, y que no tiene efectos secundarios fuera de devolver un valor.

estilo funcional de programación: Estilo de diseñar un programa en el cual la gran mayoría de los métodos son funciones puras.

modificador: Método que cambia uno o más objetos que recibe como parámetros, y generalmente devuelve `void`.

método de llenado: Tipo de método que toma un objeto “vacío” como parámetro y llena sus variables de instancia en lugar de generar un valor de retorno. Este tipo de método generalmente no es la mejor opción.

algoritmo: Conjunto de instrucciones para resolver una clase de problemas a través de un procedimiento mecánico.

9.16 Ejercicios

Ejercicio 9.1

En el juego de mesa Scrabble¹, cada ficha contiene una letra, que se usa para escribir palabras, y un puntaje, que se usa para determinar el valor de la palabra.

- Escribí una definición para la clase `Ficha` que representa las fichas del juego Scrabble. Las variables de instancia deberían ser un carácter llamado `letra` y un entero llamado `valor`.
- Escribí un constructor que toma parámetros llamados `letra` y `valor` e inicializa las variables de instancia.

1. Scrabble es una marca registrada perteneciente a Hasbro Inc. en EE.UU. y Canadá, y en el resto del mundo a J.W. Spear & Sons Limited de Maidenhead, Berkshire, Inglaterra, una subsidiaria de Mattel Inc.

- c. Escribí un método llamado `imprimirFicha` que toma un objeto `Ficha` como parámetro e imprime sus variables de instancia en algún formato fácil de leer.
- d. Escribí un método llamado `probarFicha` que crea un objeto `ficha` con la letra `Z` y el valor `10`, y luego usa `imprimirFicha` para imprimir el estado del objeto.

El objetivo de este ejercicio es practicar la parte mecánica de crear una nueva definición de clase y el código que la prueba.

Ejercicio 9.2

Escribí una definición de clase para `Fecha`, un tipo de objeto que contiene tres enteros, `anio`², `mes` y `dia`. Esta clase debe proveer dos constructores. El primero no debe tomar parámetros. El segundo debería tomar parámetros llamados `anio`, `mes` y `dia`, y usarlos para inicializar las variables de instancia.

Añadir código al `main` que crea un nuevo objeto `Fecha` llamado `nacimiento`. Este objeto debería contener tu fecha de nacimiento. Podés utilizar cualquiera de los dos constructores.

Ejercicio 9.3

Un número racional es un número que puede ser representado como el cociente entre dos enteros. Por ejemplo, $2/3$ es un número racional, y podés pensar a `7` como un número racional que tiene un `1` implícito en el denominador. Para esta tarea, vas a escribir una definición de clase para números racionales.

- a. Examiná el siguiente programa y asegurate de entender lo que hace:

```
public class Complejo {
    double real, imag;

    // constructor sencillo
    public Complejo () {
        this.real = 0.0; this.imag = 0.0;
    }

    // constructor que toma parámetros
    public Complejo (double real, double imag) {
        this.real = real; this.imag = imag;
    }

    public static void imprimirComplejo (Complejo c) {
        System.out.println (c.real + " + " + c.imag + "i");
    }
}
```

2. N.d.T.: Ni la “ñ”, ni los caracteres acentuados son válidos para nombrar nada en Java.

```

// conjugado es un modificador
public static void conjugado(Complejo c) {
    c.imag = -c.imag;
}

// abs es una función que devuelve un valor primitivo
public static double abs (Complejo c) {
    return Math.sqrt (c.real * c.real + c.imag * c.imag);
}

// sumar es una función que devuelve un nuevo Complejo
public static Complejo suma (Complejo a, Complejo b) {
    return new Complejo (a.real + b.real, a.imag + b.imag);
}

public static void main(String args[]) {

    // usa el primer constructor
    Complejo x = new Complejo ();
    x.real = 1.0;
    x.imag = 2.0;

    // usa el segundo constructor
    Complejo y = new Complejo (3.0, 4.0);

    System.out.println (Complejo.abs (y));

    Complejo.conjugado (x);
    Complejo.imprimirComplejo (x);
    Complejo.imprimirComplejo (y);

    Complejo s = Complejo.suma (x, y);
    Complejo.imprimirComplejo (s);
}
}

```

- b. Crear un nuevo programa llamado `Racional.java` que define una nueva clase llamada `Racional`. Un objeto `Racional` debe tener dos variables de instancia enteras para almacenar el numerador y el denominador de un número racional.
- c. Escribí un constructor que no toma parámetros y establece las dos variables de instancia a 0.
- d. Escribí un método llamado `imprimirRacional` que toma un objeto `Racional` como parámetro y lo imprime en algún formato razonable.

- e. Escribí un `main` que crea un nuevo objeto de tipo `Racional`, le da valores a sus variables de instancia, y finalmente imprime el objeto.
- f. En esta etapa, tenés un programa mínimo que se puede probar (depurar). Probalo, y si hace falta, depuralo.
- g. Escribí un segundo constructor que toma dos parámetros y los usa para inicializar las variables de instancia.
- h. Escribí un método llamado `invertirSigno` que invierte el signo del número racional. Este método debe ser un modificador, con lo cual debe devolver `void`. Añadí líneas al `main` para probar este método.
- i. Escribí un método llamado `invertir` que invierte el número intercambiando el numerador y el denominador. Recordá el patrón para intercambiar que vimos con anterioridad. Agregá líneas al `main` para probar este nuevo método.
- j. Escribí un método llamado `aDouble` que convierte el número racional en un `double` (número de punto flotante) y devuelve el resultado. Este método es una función pura; no modifica el objeto. Como siempre, probá el nuevo método.
- k. Escribí un modificador llamado `reducir` que reduce el número racional a sus términos más chicos. Para esto buscar el MCD del numerador y el denominador y luego dividir numerador y denominador por su MCD. Este método debe ser una función pura; no debe modificar las variables de instancia del objeto sobre el cual se llama.

Podés necesitar escribir un método llamado `mcd` que encuentra el máximo común divisor del numerador y el denominador (ver Ejercicio 5.11).

- l. Escribí un método llamado `suma` que toma dos números racionales como parámetros y devuelve un nuevo objeto `Racional`. El objeto devuelto, como es de esperarse, debe contener la suma de los parámetros.

Hay varias maneras de sumar fracciones. Podés utilizar la que quieras, pero debés asegurarte que el resultado de la operación sea reducido, de modo que el numerador y el denominador no tienen divisor común (más allá del 1).

El objetivo de este ejercicio es el de escribir una definición de clase que incluya una variedad de métodos, incluyendo constructores, modificadores y funciones puras.

Capítulo 10

Arreglos

Un **arreglo** es un conjunto de valores en el que cada valor es identificado por un índice. Se pueden crear arreglos de ints, doubles, o cualquier otro tipo, pero todos los valores en el arreglo deben tener el mismo tipo.

Sintácticamente, los tipos arreglo se ven como cualquier otro tipo de Java, excepto porque son seguidos por []. Por ejemplo, int[] es el tipo “arreglo de enteros” y double[] es el tipo “arreglo de doubles”.

Podés declarar variables con estos tipos del modo usual:

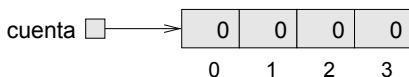
```
int[] cuenta;  
double[] valores;
```

Hasta que inicialices estas variables, estarán en null. Para crear el arreglo propiamente dicho, hay que usar el comando new.

```
cuenta = new int[4];  
valores = new double[tam];
```

La primera asignación hace que cuenta haga referencia a un arreglo de 4 enteros; la segunda hace que valores haga referencia a un arreglo de doubles. La cantidad de elementos en valores depende de tam. Podés usar cualquier expresión entera para el tamaño de un arreglo.

La siguiente figura muestra cómo son representados los arreglos en diagramas de estado:



Los números dentro de las cajas son los **elementos** del arreglo. Los números debajo de las cajas son los índices usados para identificar cada caja. Cuando se crea un nuevo arreglo, los elementos son inicializados en cero.

10.1 Accediendo a los elementos

Para almacenar valores en el arreglo, usá el operador `[]`. Por ejemplo `cuenta[0]` se refiere al “cero-ésimo” elemento del arreglo y `cuenta[1]` se refiere al “un-ésimo” elemento. Se puede usar el operador `[]` en cualquier lugar de una expresión:

```
cuenta[0] = 7;
cuenta[1] = cuenta[0] * 2;
cuenta[2]++;
cuenta[3] -= 60;
```

Todas estas son sentencias de asignación válidas. Aquí está el efecto de este fragmento de código:



A esta altura deberías haber notado que los cuatro elementos de este arreglo están numerados de 0 a 3, lo que significa que no hay un elemento con el índice 4. Esto debe sonar familiar, dado que vimos lo mismo con índices de Strings. Sin embargo, es un error común el irse más allá de los bordes de un arreglo, lo que causa una `ArrayIndexOutOfBoundsException` (excepción de arreglo fuera de límites). Como con todas las excepciones, sale un mensaje de error y el programa termina.

Podés usar cualquier expresión como un índice, siempre y cuando sea de tipo `int`. Una de las maneras más comunes de indexar un arreglo es con una variable de ciclo. Por ejemplo:

```
int i = 0;
while (i < 4) {
    System.out.println (cuenta[i]);
    i++;
}
```

Este es un ciclo `while` estándar que cuenta de 0 hasta 4, y cuando la variable `i` del ciclo es 4, la condición se hace falsa y el ciclo termina. Por lo tanto, el cuerpo del ciclo solo se ejecuta cuando `i` es 0, 1, 2 y 3.

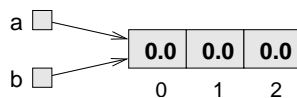
En cada iteración del ciclo usamos *i* como un índice del arreglo e imprimimos el *i*-ésimo elemento. Este tipo de recorrida por un arreglo es muy común. Los arreglos y ciclos quedan bien juntos como el maní con una buena cerveza.

10.2 Copiando arreglos

Cuando copiás una variable arreglo, no olvides que estás copiando una referencia al arreglo. Por ejemplo:

```
double[] a = new double [3];  
double[] b = a;
```

Este código crea un arreglo de tres doubles, y guarda dos referencias a él en dos variables distintas. Esta situación es una forma de aliasing.



Un cambio en cualquier arreglo se verá reflejado en el otro. Este no es el comportamiento que usualmente se quiere; en cambio, deberías hacer una copia del arreglo, creando uno nuevo y copiando cada elemento de uno en el otro.

```
double[] b = new double [3];  
  
int i = 0;  
while (i < 4) {  
    b[i] = a[i];  
    i++;  
}
```

10.3 Ciclos for

Los ciclos que hemos escrito hasta ahora tienen una cierta cantidad de elementos en común. Todos ellos comienzan inicializando una variable; tienen una condición, o test, que depende de esa variable; y dentro del ciclo hacen algo con esa variable, como incrementarla.

Este ciclo es tan común que existe una sentencia alternativa de ciclos, llamada *for* (del inglés, para), que lo expresa de manera más concisa. La sintaxis general se ve así:

```
for (INICIALIZADOR; CONDICION; INCREMENTADOR) {  
    CUERPO  
}
```

Esta sentencia es exactamente equivalente a

```
INICIALIZADOR;  
while (CONDICION) {  
    CUERPO  
    INCREMENTADOR  
}
```

excepto porque es más concisa y, dado que concentra todas las sentencias relacionadas a un ciclo en un solo lugar, es más fácil de leer. Por ejemplo:

```
for (int i = 0; i < 4; i++) {  
    System.out.println (cuenta[i]);  
}
```

es equivalente a

```
int i = 0;  
while (i < 4) {  
    System.out.println (cuenta[i]);  
    i++;  
}
```

Como ejercicio, escribí un ciclo for para copiar los elementos de un arreglo.

10.4 Arreglos y objetos

Bajo muchos puntos de vista, los arreglos se comportan como objetos:

- Cuando se declara una variable arreglo, obtenés una referencia al arreglo.
- Hay que usar el comando new para crear el arreglo propiamente dicho.
- Cuando se pasa un arreglo como parámetro, se pasa una referencia, lo que quiere decir que el método llamado puede cambiar los contenidos del arreglo.

Algunos de los objetos que hemos visto, como los Rectangulos, son similares a los arreglos, en el sentido de que son colecciones de valores con nombre. Esto trae a colación la pregunta, “¿En qué se diferencia un arreglo de 4 enteros de un objeto Rectangle?”

Si volvemos a la definición de “arreglo” al comienzo del capítulo, vamos a encontrar una diferencia, que es que los elementos de un arreglo son identificados por índices, mientras que los elementos (variables de instancia) de un objeto tienen nombres (como x, ancho, etc.).

Otra diferencia entre arreglos y objetos es que todos los elementos de un arreglo tienen que ser del mismo tipo. Aunque eso también sea cierto para Rectangulos, hemos visto otros objetos que tienen variables de instancia de diferentes tipos (como Tiempo).

10.5 Longitud de un arreglo

En realidad, los arreglos sí tienen una variable de instancia con nombre: `length`¹. Para sorpresa de nadie, contiene la longitud del arreglo (su cantidad de elementos). Es buena idea usar este valor como el borde superior de un ciclo, en lugar de un valor constante. De esta manera, si el tamaño del arreglo cambia, no tendrás que revisar el programa cambiando todos los ciclos; funcionarán correctamente para cualquier tamaño de arreglo.

```
for (int i = 0; i < a.length; i++) {  
    b[i] = a[i];  
}
```

La última vez que el cuerpo del ciclo se ejecuta, `i` vale `a.length-1`, que es el índice del último elemento. Cuando `i` es igual a `a.length`, la condición falla y el cuerpo no es ejecutado, lo que es bueno, pues causaría una excepción. Este código asume que el arreglo `b` contiene al menos tantos elementos como `a`.

Como ejercicio, escribí un método llamado `clonarArreglo` que toma un arreglo de enteros como parámetro, crea un nuevo arreglo del mismo tamaño, copia los elementos del primer arreglo al nuevo, y devuelve una referencia al nuevo arreglo.

10.6 Números aleatorios

La mayoría de los programas hacen lo mismo cada vez que son ejecutados, por lo que se dice que son **determinísticos**. Usualmente, el deter-

1. N.d.T.: Longitud.

minismo es algo bueno, pues esperamos que el mismo cálculo dé el mismo resultado. Para algunas aplicaciones, sin embargo, nos gustaría que la computadora fuera impredecible. Los juegos son un ejemplo obvio, pero hay muchos más.

Hacer un programa verdaderamente **no determinístico** resulta no ser tan sencillo, pero hay maneras de hacer que, al menos, parezca no determinístico. Una de ellas es generando números aleatorios y usándolos para determinar el resultado de un programa. Java provee un método preincorporado que genera números **pseudoaleatorios**, que no son realmente aleatorios en sentido matemático, pero para nuestros propósitos, serán suficientes.

Revisá la documentación del método `random` en la clase `Math`. El valor de retorno es un `double` entre 0.0 y 1.0. Para ser precisos, es mayor o igual que 0.0 y menor estricto que 1.0. Cada vez que llames a `random` obtenés el siguiente número en una secuencia pseudoaleatoria. Para ver un ejemplo, corré este ciclo:

```
for (int i = 0; i < 10; i++) {  
    double x = Math.random ();  
    System.out.println (x);  
}
```

Para generar un `double` aleatorio entre 0.0 y una cota superior como `alto`, podés multiplicar `x` por `alto`. ¿Cómo harías para generar un número aleatorio entre `bajo` y `alto`?

Ejercicio 10.1

Escribí un método llamado `doubleAleatorio` que tome dos `doubles`, `alto` y `bajo`, y que devuelva un `double` aleatorio x de modo tal que $bajo \leq x < alto$.

Ejercicio 10.2

Escribí un método llamado `enteroAleatorio` que tome dos parámetros, `bajo` y `alto`, y que devuelva un entero entre `bajo` y `alto` (incluyendo a ambos).

10.7 Arreglos de números aleatorios

Si tu implementación de `enteroAleatorio` es correcta, entonces cada valor en el rango entre `bajo` y `alto` debería tener la misma probabilidad. Si generases una larga serie de números, cada valor debería aparecer, al menos aproximadamente, la misma cantidad de veces.

Una forma de testear tu método es generar una gran cantidad de valores aleatorios, guardarlos en un arreglo, y contar la cantidad de veces que ocurre cada valor.

El siguiente método toma un solo argumento, el tamaño del arreglo. Crea un nuevo arreglo de enteros, lo llena con valores aleatorios, y devuelve una referencia al nuevo arreglo.

```
public static int[] arregloAleatorio (int n) {  
    int[] a = new int[n];  
    for (int i = 0; i<a.length; i++) {  
        a[i] = enteroAleatorio (0, 100);  
    }  
    return a;  
}
```

El tipo de retorno es `int[]`, lo que significa que este método devuelve un arreglo de enteros. Para testear este método, es conveniente tener un método que imprima el contenido de un arreglo.

```
public static void imprimirArreglo (int[] a) {  
    for (int i = 0; i<a.length; i++) {  
        System.out.println (a[i]);  
    }  
}
```

El siguiente código genera un arreglo y lo imprime:

```
int cantValores = 8;  
int[] arreglo = arregloAleatorio (cantValores);  
imprimirArreglo (arreglo);
```

En mi máquina la salida es

```
27  
6  
54  
62  
54  
2  
44  
81
```

que se ve bastante aleatorio. Tus resultados pueden diferir.

Si estas fueran notas de un examen, unas bastante malas, el profesor puede presentar los resultados a la clase en forma de **histograma**, que es un conjunto de contadores que almacena la cantidad de veces que aparece cada valor.

Para notas de exámenes, podríamos tener diez contadores para ir llevando la cuenta de cuántos estudiantes se sacaron notas en los 90s, los 80s, etc. Las siguientes secciones desarrollan código para generar un histograma.

10.8 Contando

Un buen acercamiento a problemas como este es pensar en métodos simples que sean fáciles de escribir, y que tal vez resulten útiles. Entonces se los puede combinar en una solución. Claro que no es fácil saber de antemano qué métodos serán probablemente útiles, pero a medida que ganes experiencia, tendrás una mejor idea.

Además, no es siempre obvio qué clase de cosas son fáciles de escribir, pero un buen acercamiento es ir en busca de subproblemas que se acomoden a patrones que hayas visto antes.

En la sección 7.7 vimos un ciclo que recorría una cadena y contaba el número de veces que aparecía una letra dada. Podés pensar a este programa como un ejemplo de un patrón llamado “recorrer y contar.” Los elementos de este patrón son:

- Un conjunto o contenedor que pueda ser recorrido, como un arreglo o una cadena.
- Un test que pueda ser aplicado a cada elemento en el contenedor.
- Un contador que lleve la cuenta de cuántos elementos aprueban el test.

En este caso, el contenedor es un arreglo de enteros. El test es si una nota dada cae o no en un rango de valores dado.

Aquí, un método llamado `enRango` que cuenta la cantidad de elementos de un arreglo que caen en un rango dado. Los parámetros son el arreglo y dos enteros que especifican los bordes inferior y superior del rango.

```
public static int enRango (int[] a, int bajo, int alto) {  
    int contador = 0;  
    for (int i=0; i<a.length; i++) {  
        if (a[i] >= bajo && a[i] < alto)  
            contador++;  
    }  
    return count;  
}
```

En mi descripción del método, no fui muy cuidadoso con respecto a si algo igual a bajo o alto cae en el rango, pero podrás ver en el código que bajo

está adentro y alto está afuera. Eso debería garantizarnos que no vamos a contar ningún elemento dos veces.

Ahora podemos contar la cantidad de notas en los rangos que nos interesan:

```
int[] notas = arregloAleatorio (30);
int a = enRango (notas, 90, 100);
int b = enRango (notas, 80, 90);
int c = enRango (notas, 70, 80);
int d = enRango (notas, 60, 70);
int f = enRango (notas, 0, 60);
```

10.9 El histograma

El código que tenemos hasta ahora es un poco repetitivo, pero es aceptable siempre y cuando la cantidad de rangos que nos interesen sea pequeña. Pero ahora imaginemos que queremos llevar la cuenta de la cantidad de veces que aparece cada nota, los 100 posibles valores. Te gustaría escribir:

```
int cuenta0 = enRango (notas, 0, 1);
int cuenta1 = enRango (notas, 1, 2);
int cuenta2 = enRango (notas, 2, 3);
...
int cuenta99 = enRango (notas, 99, 100);
```

No lo creo. Lo que realmente queremos es una manera de almacenar 100 enteros, preferiblemente para poder usar un índice para acceder a cada uno. Inmediatamente, deberías estar pensando en ¡un arreglo!

El patrón de contar es el mismo si usamos un solo contador o un arreglo de contadores. En este caso, inicializamos el arreglo afuera del ciclo; luego, dentro del ciclo, llamamos a `enRango` y almacenamos el resultado:

```
int[] cuentas = new int [100];

for (int i = 0; i<100; i++) {
    cuentas[i] = enRango (notas, i, i+1);
}
```

El único truquito aquí es que estamos usando la variable del ciclo en dos roles: como un índice en un arreglo, y como parámetro para `enRango`.

10.10 Solución de una sola pasada

Aunque este código funciona, no es tan eficiente como podría serlo. Cada vez que llama a `enRango`, atraviesa el arreglo entero. A medida que el número de rangos aumenta, terminan siendo muchas recorridas.

Sería mejor hacer una sola pasada por el arreglo, y para cada valor, computar en qué rango cae. Entonces podríamos incrementar el contador apropiado. En este ejemplo, ese cálculo es trivial, porque podemos usar el mismo valor como índice en el arreglo de contadores.

Aquí está un código que recorre un arreglo de notas, una única vez, y genera un histograma.

```
int[] cuentas = new int [100];

for (int i = 0; i < notas.length; i++) {
    int indice = notas[i];
    cuentas[indice]++;
}
```

Ejercicio 10.3

Encapsulá este código en un método llamado `histogramaNotas` que tome un arreglo de notas y devuelva un histograma de los valores del arreglo.

Modificá el método de forma tal que el histograma tenga solamente 10 contadores, y cuente la cantidad de notas en cada rango de 10 valores; eso es, los 90s, los 80s, etc.

10.11 Glosario

arreglo: Colección nombrada de valores, en donde todos los valores son del mismo tipo, y cada valor es identificado por un índice.

colección: Cualquier estructura de datos que contenga un conjunto de ítems o elementos.

elemento: Uno de los valores de un arreglo. El operador `[]` selecciona elementos de un arreglo.

índice: Variable o valor entero usado para indicar un elemento en un arreglo.

determinístico: Programa que hace lo mismo cada vez que es llamado.

pseudoaleatorio: Secuencia de números que parecen ser aleatorios, pero que son en realidad el resultado de una computación determinística.

histograma: Arreglo de enteros en el que cada entero cuenta la cantidad de valores que caen dentro de un cierto rango.

10.12 Ejercicios

Ejercicio 10.4

Escribí un método de clase llamado `sonFactores` que tome un entero `n` y un arreglo de enteros, y devuelva `true` si los números del arreglo son factores de `n` (es decir, si `n` es divisible por todos ellos). PISTA: Ver Ejercicio 5.2.

Ejercicio 10.5

Escribí un método que tome un arreglo de enteros y un entero llamado `blanco` como parámetros, y devuelva el primer índice donde `blanco` aparece en el arreglo, si lo hace, y `-1` en caso contrario.

Ejercicio 10.6

Escribí un método llamado `histogramaArreglo` que tome un arreglo de enteros y devuelva un nuevo arreglo `histograma`. El `histograma` debería contener 11 elementos con los siguientes contenidos:

```
elemento 0 -- cantidad de elementos en el arreglo que son <= 0
          1 -- cantidad de elementos en el arreglo que son == 1
          2 -- cantidad de elementos en el arreglo que son == 2
          ...
          9 -- cantidad de elementos en el arreglo que son == 9
         10 -- cantidad de elementos en el arreglo que son >= 10
```

Ejercicio 10.7

Algunos programadores disienten con la regla general de que debe dárseles nombres declarativos a las variables y métodos. En cambio, piensan que las variables y los métodos deben llevar nombres de frutas. Para cada uno de los siguientes métodos, escribí una sentencia que describa abstractamente qué hace el método. Para cada variable, identificá el papel que juega.

```
public static int banana (int[] a) {
    int uva = 0;
    int i = 0;
    while (i < a.length) {
        uva = uva + a[i];
        i++;
    }
}
```

```

        return uva;
    }

    public static int manzana (int[] a, int p) {
        int i = 0;
        int pera = 0;
        while (i < a.length) {
            if (a[i] == p) pera++;
            i++;
        }
        return pera;
    }

    public static int pomelo (int[] a, int p) {
        for (int i = 0; i<a.length; i++) {
            if (a[i] == p) return i;
        }
        return -1;
    }
}

```

El objetivo de este ejercicio es practicar la lectura de código y el reconocimiento de los patrones de solución que hemos visto.

Ejercicio 10.8

- ¿Cuál es la salida del siguiente programa?
- Dibujá un diagrama de pila que muestre el estado del programa justo antes de la sentencia `return` de `mus`.
- Describí en pocas palabras lo que hace `mus`.

```

public static int[] hacer (int n) {
    int[] a = new int[n];

    for (int i=0; i<n; i++) {
        a[i] = i+1;
    }
    return a;
}

public static void dub (int[] jub) {
    for (int i=0; i<jub.length; i++) {
        jub[i] *= 2;
    }
}

```

```

public static int mus (int[] zoo) {
    int fus = 0;
    for (int i=0; i<zoo.length; i++) {
        fus = fus + zoo[i];
    }
    return fus;
}

public static void main (String[] args) {
    int[] bruno = hacer (5);
    dub (bruno);

    System.out.println (mus (bruno));
}

```

Ejercicio 10.9

Muchos de los patrones que hemos visto para recorrer arreglos también pueden ser escritos recursivamente. No es común hacerlo así, pero es bueno como ejercicio.

- a. Escribí un método llamado `maximoEnRango` que tome un arreglo de enteros y un rango de índices (`indiceBajo` e `indiceAlto`), que encuentre el máximo valor del arreglo, considerando solamente los elementos entre `indiceBajo` e `indiceAlto`, incluyendo ambos extremos.

Este método debería ser recursivo. Si la longitud del rango es 1, es decir, si `indiceBajo == indiceAlto`, sabemos inmediatamente que el único elemento en el rango es el máximo. Entonces ese es el caso base.

Si hubiera más de un elemento en el rango, podemos partir el arreglo en dos, encontrar el máximo en cada parte, y luego encontrar el máximo entre los dos máximos parciales.

- b. Métodos como `maximoEnRango` pueden ser incómodos de usar. Para encontrar el máximo de un arreglo, tenemos que proveerle el rango que incluya al arreglo entero.

```
double max = maxEnRango (arreglo, 0, a.length-1);
```

Escribí un método llamado `max` que tome un arreglo como parámetro y que use a `maximoEnRango` para encontrar y devolver al valor más alto. Métodos como `max` son algunas veces llamados **métodos encapsuladores** (o `wrappers`) porque proveen una capa de abstracción alrededor de un método incómodo de usar y provee una interfaz para el mundo exterior para que sea más fácil de usar. El método que realmente realiza los cálculos es llamado **método auxiliar** (o `helper`). Veremos este patrón de nuevo en la Sección 14.9.

- c. Escribí una versión recursiva de encontrar usando el patrón encapsulador-auxiliar. encontrar debería tomar un arreglo de enteros y un entero objetivo. Debería devolver el índice de la primera posición en la que el entero objetivo aparece en el arreglo, o -1 en caso de que no aparezca.

Ejercicio 10.10

Una manera no muy eficiente de ordenar los elementos de un arreglo es encontrar el mayor e intercambiarlo con el primero, después encontrar el segundo mayor, e intercambiarlo con el segundo, y así siguiendo.

- a. Escribí un método llamado `indiceDeMaximoEnRango` que tome un arreglo de enteros, encuentre el mayor elemento en el rango dado, y devuelva *su índice*. Podés modificar tu versión recursiva de `maximoEnRango` o empezar una versión iterativa desde cero.
- b. Escribí un método llamado `intercambiarElemento` que tome un arreglo de enteros y dos índices, y que intercambie los elementos de los índices dados.
- c. Escribí un método llamado `ordenarArreglo` que tome un arreglo de enteros y que use `indiceDeMaximoEnRango` e `intercambiarElemento` para ordenar el arreglo de mayor a menor.

Ejercicio 10.11

Escribí un método llamado `histogramaLetras` que tome una `String` como parámetro y devuelva un histograma de las letras del `String`. El cero-ésimo elemento del histograma deber contener la cantidad de a's en la `String` (mayúsculas y minúsculas); el 25to elemento debe contener la cantidad de z's. Tu solución debe recorrer el `String` una sola vez.

Ejercicio 10.12

Se dice que una palabra es una “redoblona” si cada letra de ella aparece exactamente dos veces. Por ejemplo, las siguientes palabras son algunas de las redoblonas que encontré en mi diccionario.

coco, crecer, dada, fofo, insistente, narran, osos, papa, rara, tratar.

Escribí un método llamado `esRedoblona` que devuelva `true` si recibe una palabra redoblona y `false` de lo contrario.

Ejercicio 10.13

En el Scrabble, cada jugador tiene un conjunto de fichas con letras, y el objetivo del juego es usar esas letras para formar palabras. El sistema de puntuación es complicado, pero como guía a groso modo, las palabras más largas frecuentemente valen más que las cortas.

Imaginá que te dan un conjunto de fichas representado con un String, por ejemplo "qi jibo" y te dan otro String para testear, por ejemplo, "jib". Escribí un método llamado `testearPalabra` que tome estas dos Strings y devuelva verdadero si el conjunto de fichas puede ser usado para deletrear la palabra. Podés tener más de una ficha con la misma letra, pero cada ficha puede ser usada una sola vez.

Ejercicio 10.14

En el verdadero Scrabble, hay algunas fichas en blanco que pueden ser usadas como comodines; es decir, se pueden usar para representar cualquier letra.

Pensá un algoritmo para `testearPalabra` que trabaje con comodines. No te estanques con detalles de implementación como ver de qué forma representar comodines. Simplemente describí el algoritmo, usando el castellano, pseudocódigo, o Java.

Capítulo 11

Arreglos de Objetos

11.1 Composición

Hasta ahora hemos visto varios ejemplos de composición (la capacidad de combinar herramientas del lenguaje en una variedad de formas). Uno de los primeros ejemplos que vimos fue el uso de la llamada a un método como parte de una expresión. Otro ejemplo es la estructura anidada de sentencias: podés poner una sentencia `if` dentro de un ciclo `while`, o dentro de otra sentencia `if`, etc.

Habiendo visto estos patrones, y habiendo aprendido de arreglos y objetos, no debería sorprenderte el hecho de que puedas tener arreglos de objetos. De hecho, podés incluso tener objetos que contengan arreglos (como variables de instancia); podés tener arreglos que contengan arreglos; podés tener objetos que contengan objetos, y así sucesivamente.

En los próximos dos capítulos veremos algunos ejemplos de estas combinaciones, usando objetos de tipo `Carta` como ejemplo.

11.2 Objetos `Carta`

Si no estás familiarizado con la baraja de cartas francesas, este sería un buen momento para agarrar un mazo, si no este capítulo no va a tener mucho sentido. Hay 52 cartas en un mazo, cada una de ellas pertenece a uno de los cuatro *palos* y tiene uno de 13 valores. Los palos son Picas, Corazones, Diamantes y Tréboles (en orden descendiente para el juego `Bridge`). Los valores son As, 2, 3, 4, 5, 6, 7, 8, 9, 10, J (Jota), Q (Reina) y

K (Rey)¹. Dependiendo de qué juego estés jugando, el valor del As puede ser superior al Rey o inferior al 2.

Si queremos definir un nuevo objeto para representar una carta de la baraja, es bastante obvio cuáles deberían ser las variables de instancia: **valor** y **palo**. Lo que no es tan obvio es de qué tipo deberían ser estas variables de instancia. Una posibilidad es usar `Strings`, con cosas como "Pica" para los palos y "Q" para los valores. Un problema de esta implementación es que no será fácil comparar las cartas para ver cuál tiene mayor valor o palo.

Una alternativa es usar enteros para **codificar** los valores y los palos. Con "codificar," no me refiero a lo que algunas personas creen, que es encriptar, o traducir a un código secreto. Lo que un científico de la computación quiere decir con "codificar" es algo como "definir una relación entre una secuencia de números y las cosas que quiero representar." Por ejemplo,

Picas	\mapsto	3
Corazones	\mapsto	2
Diamantes	\mapsto	1
Tréboles	\mapsto	0

La utilidad obvia de esta relación (o *mapeo*²) es que los palos están relacionados con enteros ordenados, y por lo tanto podemos compararlos comparando los enteros. El mapeo para los valores es bastante obvio; cada valor numérico se mapea con su correspondiente entero, y para las cartas con figuras:

J	\mapsto	11
Q	\mapsto	12
K	\mapsto	13

La razón por la cual uso notación matemática para estos mapeos es que no son parte del programa en Java. Son parte del diseño del programa, pero nunca aparecen de manera explícita en el código. La definición de la clase para el tipo `Carta` se ve así:

1. N.d.T.: Estas últimas tres cartas en inglés son: Jack, Queen y King; de ahí las letras J, Q y K.

2. N.d.T.: De la palabra inglesa *mapping* que significa relación, no del verbo castellano *mapear*, el cual significa dibujar mapas.

```

class Carta
{
    int palo, valor;

    public Carta () {
        this.palo = 0;  this.valor = 0;
    }

    public Carta (int palo, int valor) {
        this.palo = palo;  this.valor = valor;
    }
}

```

Como de costumbre, proveo dos constructores, uno de los cuales toma un parámetro para cada variable de instancia y otro que no toma parámetros.

Para crear un objeto que represente el 3 de trébol, debemos usar el comando new:

```

Carta tresDeTrebol = new Carta (0, 3);

```

El primer argumento, 0 representa el palo Tréboles.

11.3 El método imprimirCarta

Cuando creás una nueva clase, el primer paso es usualmente declarar las variables de instancia y escribir los constructores. El segundo paso generalmente es escribir los métodos estándar que todo objeto debe tener, incluyendo uno que imprima el objeto y uno o dos que comparen objetos. Voy a empezar con imprimirCarta.

Para imprimir objetos de tipo Carta de forma tal que el humano pueda leer fácilmente, queremos mapear los códigos numéricos en palabras. Una forma natural de hacer eso es mediante un arreglo de Strings. Podés crear un arreglo de Strings de la misma manera en la que creás un arreglo de tipos primitivos:

```

String[] palos = new String [4];

```

Luego, podemos setear los valores de los elementos del arreglo.

```

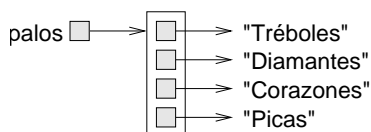
palos[0] = "Tréboles";
palos[1] = "Diamantes";
palos[2] = "Corazones";
palos[3] = "Picas";

```

Crear un arreglo e inicializar sus elementos es algo tan común que Java provee una sintaxis especial para ello:

```
String[] palos = { "Tréboles", "Diamantes",  
                  "Corazones", "Picas" };
```

El efecto de esta sentencia es idéntica a las de declaración y asignación. Un diagrama de estado de este arreglo sería algo como:



Los elementos del arreglo son *referencias* a los Strings y no los Strings mismos. Esto es así para cualquier arreglo de objetos, lo veremos en detalle más adelante. Por ahora, todo lo que necesitamos es otro arreglo de Strings para decodificar los valores:

```
String[] valores = { "nada", "As", "2", "3", "4", "5",  
                    "6", "7", "8", "9", "10", "J", "Q", "K" };
```

El motivo del "nada" es el de ocupar la posición cero del arreglo, la cual no se usará nunca. Los únicos valores válidos son del 1 al 13. Claro que este elemento desperdiciado no es necesario. Podríamos haber empezado por el 0, como de costumbre, pero es mejor codificar el 2 con el 2, el 3 con el 3, etc.

Usando estos arreglos, podemos seleccionar los Strings apropiadamente usando como índices los valores de palo y valor. En el método `imprimirCarta`,

```
public static void imprimirCarta (Carta c) {  
    String[] palos = { "Tréboles", "Diamantes",  
                      "Corazones", "Picas" };  
    String[] valores = { "nada", "As", "2", "3", "4", "5",  
                        "6", "7", "8", "9", "10", "J", "Q", "K" };  
  
    System.out.println (valores[c.valor] + " de " + palos[c.palo]);  
}
```

la expresión `palos[c.palo]` significa "usó la variable de instancia `palo` del objeto `c` como índice en el arreglo llamado `palos`, y seleccionó la cadena apropiada". La salida del siguiente código

```
Carta carta = new Carta (1, 11);  
imprimirCarta(carta);
```

es J de Diamantes.

11.4 El método `mismaCarta`

La palabra “misma” es una de esas cosas del lenguaje natural que parecen perfectamente claras hasta que las pensás un poco más en profundidad y te das cuenta que es más de lo que realmente pensabas.

Por ejemplo, si yo digo “Cristian y yo tenemos el mismo auto”, quiero decir que su auto y el mío son de la misma marca y modelo, pero son dos autos diferentes. Si digo “Cristian y yo tenemos la misma madre”, quiero decir que su madre y la mía son una y la misma. Entonces la idea de ser el “mismo” difiere dependiendo del contexto.

Cuando hablamos de objetos, hay una ambigüedad similar. Por ejemplo, si dos Cartas son la misma, ¿quiere decir que tienen los mismos valores (palo y valor) o son realmente el mismo objeto Carta?

Para ver si dos referencias apuntan al mismo objeto, podemos usar el operador de comparación `==`. Por ejemplo:

```
Carta carta1 = new Carta (1, 11);  
Carta carta2 = carta1;  
  
if (carta1 == carta2) {  
    System.out.println ("carta1 y carta2 son el mismo objeto.");  
}
```

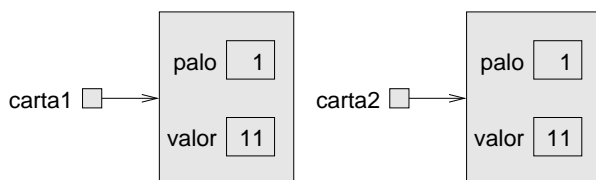
Este tipo de igualdad se llama **igualdad superficial** porque sólo compara las referencias, no el contenido de los objetos. Para comparar el contenido de los objetos—**igualdad de contenido**—es común escribir un método con un nombre como `mismaCarta`.

```
public static boolean mismaCarta (Carta c1, Carta c2) {  
    return (c1.palo == c2.palo && c1.valor == c2.valor);  
}
```

Así, si creamos dos objetos diferentes que contienen la misma información, podemos usar `mismaCarta` para ver si representan la misma carta:

```
Carta carta1 = new Carta (1, 11);  
Carta carta2 = new Carta (1, 11);  
  
if (mismaCarta (carta1, carta2)) {  
    System.out.println ("carta1 y carta2 son la misma carta.");  
}
```

En este caso, `carta1` y `carta2` son dos objetos diferentes que contienen la misma información



y así la condición es verdadera. ¿Cómo se ve el diagrama de estado cuando `carta1 == carta2` es verdadero?

En la Sección 7.10 dije que no se debe usar el operador `==` en `Strings` porque no hace lo que uno espera. En lugar de comparar los contenidos del `String` (igualdad de contenidos), verifica si los dos `Strings` son el mismo objeto (igualdad superficial).

11.5 El método `compararCarta`

Para los tipos primitivos, existen operadores condicionales que comparan valores y determinan cuándo uno es mayor o menor que otro. Estos operadores (`<`, `>` y otros) no funcionan para objetos. Para `Strings` existe un método preincorporado llamado `compareTo`³. Para `Cartas` tenemos que escribir nuestro propio método, el cual llamaremos `compararCarta`. Más tarde usaremos este método para ordenar un mazo de cartas.

Algunos conjuntos son completamente ordenables, lo que significa que podés comparar cualquier par de elementos y decir cuál es más grande. Por ejemplo, los números enteros y los de punto flotante están totalmente ordenados. Algunos conjuntos no tienen orden, lo que significa que no tiene ningún sentido decir si un elemento es más grande que otro. Por ejemplo, las frutas no tienen orden, por lo cual no podemos comparar manzanas con naranjas. En Java, el tipo `boolean` no tiene orden; no podemos decir si `true` es mayor que `false`.

El conjunto de las cartas tiene un orden parcial, lo que indica que a veces podemos comparar cartas y a veces no. Por ejemplo, sabemos que el 3 de Tréboles es mayor que el 2 de Tréboles, y el 3 de Diamantes es mayor que el 3 de Tréboles. Pero ¿qué es mejor, el 3 de Tréboles o el 2 de Diamantes? Una tiene un valor mayor, pero la otra tiene un mayor palo.

Para hacer que las cartas sean comparables, tenemos que decidir cuál es más importante, el valor o el palo. Para ser honesto, la decisión es completamente arbitraria.

3. N.d.T.: En inglés significa “comparar con”.

Sólo para decidir esto, diremos que el palo es más importante que el valor, porque cuando comprás un mazo de cartas, viene ordenado con todos los Tréboles primero, seguido de todos los Diamantes, y así.

Con esta decisión tomada, podemos escribir `compararCarta`. Tomará dos Cartas como parámetro y devolverá 1 si la primera carta gana, -1 si la segunda gana y 0 si empatan (indicando igualdad de contenido). A veces es confuso deducir estos valores de retorno de manera directa, pero son bastante estándar para métodos de comparación.

Primero comparamos los palos:

```
if (c1.palo > c2.palo) return 1;
if (c1.palo < c2.palo) return -1;
```

Si ninguna de estas sentencias es verdadera, entonces los palos tienen que ser iguales, y tenemos que comparar los valores:

```
if (c1.valor > c2.valor) return 1;
if (c1.valor < c2.valor) return -1;
```

Si ninguna de estas es verdadera, los valores tienen que ser iguales, y entonces devolvemos 0. Con este orden, los ases serán menores que los dos.

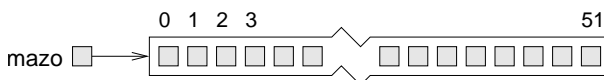
Como ejercicio, arreglalo para que los ases sean mayores que las K, y encapsulá este código en un método.

11.6 Arreglos de cartas

La razón por la cual elegí Cartas para los objetos de este capítulo es porque hay un uso obvio para un arreglo de cartas—un mazo. Acá hay un código que crea un nuevo mazo de 52 cartas:

```
Carta[] mazo = new Carta [52];
```

Acá está el diagrama de estado para este objeto:



El punto importante a ver acá es que el arreglo contiene sólo *referencias* a objetos; no contiene ningún objeto de tipo `Carta`. Los valores de los elementos del arreglo son inicializados a `null`. Podés acceder a los elementos del arreglo de forma usual:


```

if (mazo[3] == null) {
    System.out.println ("¡No hay cartas aun!");
}

```

Pero si intentás acceder a las variables de instancia de las Cartas inexistentes, vas a obtener una excepción de tipo `NullPointerException`⁴.

```

mazo[2].valor;           // NullPointerException

```

No obstante, esa es la manera correcta de acceder al valor de la segunda carta del mazo (en realidad de la tercera—empieza en cero, ¿te acordás?). Este es otro ejemplo de composición, la combinación de la sintaxis para acceder a un elemento de un arreglo y a una variable de instancia de un objeto.

La manera más fácil de llenar el mazo con objetos de tipo `Carta` es escribir un ciclo anidado:

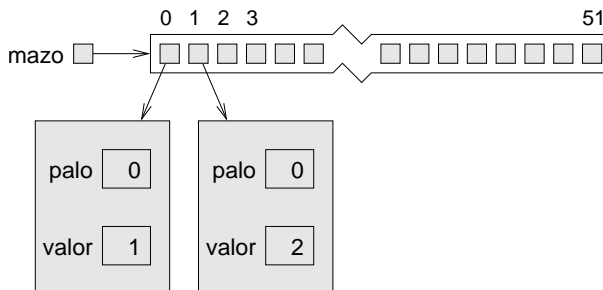
```

int indice = 0;
for (int palo = 0; palo <= 3; palo++) {
    for (int valor = 1; valor <= 13; valor++) {
        mazo[indice] = new Carta (palo, valor);
        indice++;
    }
}

```

El ciclo externo enumera los palos, de 0 a 3. Para cada palo, el ciclo interno enumera los valores, de 1 a 13. Dado que el ciclo externo itera 4 veces, y el ciclo interno itera 13 veces, el número total de veces que se ejecuta el cuerpo es 52 (4 veces 13).

Usé la variable `indice` para ir marcando el lugar en el que debería ir la siguiente carta. El siguiente diagrama de estado muestra cómo se ve el mazo después de que las primeras dos cartas fueron asignadas:



4. N.d.T.: En inglés significa “excepción por puntero a null”.

Ejercicio 11.1

Encapsulá este código que construye mazos dentro de un método llamado `construirMazo` que no tome parámetros y que devuelva un arreglo de Cartas totalmente lleno.

11.7 El método `imprimirMazo`

Siempre que estás trabajando con arreglos, es conveniente tener un método que imprima el contenido del arreglo. Ya vimos el patrón para recorrer un arreglo varias veces, con lo cual el siguiente método debería serte familiar:

```
public static void imprimirMazo (Carta[] mazo) {
    for (int i=0; i<mazo.length; i++) {
        imprimirCarta (mazo[i]);
    }
}
```

Como `mazo` es de tipo `Carta[]`, un elemento de `mazo` es de tipo `Carta`. Por lo tanto, `mazo[i]` es un argumento válido para `imprimirCarta`.

11.8 Búsqueda

El próximo método que quisiera escribir es `buscarCarta`, que busca en un arreglo de Cartas para ver si contiene una determinada carta. Puede no ser obvio para qué se podría usar este método, pero me da la oportunidad de mostrar dos maneras de buscar cosas, una búsqueda lineal y una búsqueda *binaria*.

La búsqueda lineal es la más obvia de las dos; consiste en recorrer el mazo y comparar cada carta con la carta buscada. Si la encontramos devolvemos el índice en el que aparece la carta. Si no está en el mazo, devolvemos -1.

```
public static int buscarCarta (Carta[] mazo, Carta carta) {
    for (int i = 0; i< mazo.length; i++) {
        if (mismaCarta (mazo[i], carta)) return i;
    }
    return -1;
}
```

Los argumentos de `buscarCarta` son `carta` y `mazo`. Puede parecer un poco raro tener una variable con el mismo nombre que un tipo (la variable

carta tiene tipo Carta). Esto es válido y usual, aunque a veces puede hacer que el código sea difícil de leer. En este caso, sin embargo, creo que funciona.

El método termina tan pronto encuentra la carta, lo que indica que no tenemos que recorrer el mazo entero si encontramos la carta que buscábamos. Si el ciclo termina sin encontrar la carta, sabemos que la carta no está en el mazo y entonces devolvemos -1.

Si las cartas en el mazo no están en orden, no existe una manera de buscar que sea más rápida que esta. Debemos examinar cada carta, ya que de otra manera no podemos asegurar que la carta que queremos no está en el mazo.

Pero cuando buscás una palabra en el diccionario, no buscás linealmente pasando por todas las palabras. Y esto es porque las palabras están en orden alfabético. Entonces, probablemente uses un algoritmo similar a la búsqueda binaria:

1. Empezar en algún lugar por la mitad.
2. Buscar una palabra en la página y compararla con la palabra que buscás.
3. Si encontraste la palabra buscada, parar.
4. Si la palabra que estás buscando viene antes de la palabra de la página, saltar a algún lugar más atrás en el diccionario e ir al paso 2.
5. Si la palabra que estás buscando viene después de la palabra de la página, saltar a algún lugar más adelante en el diccionario e ir al paso 2.

Si en algún momento llegás al punto en que hay dos palabras adyacentes en la página y tu palabra vendría entre medio de ellas, podés deducir que tu palabra no está en el diccionario. La única opción sería que tu palabra haya sido mal ubicada en algún otro lado, pero eso contradice nuestra suposición de que las palabras estaban todas en orden alfabético.

En el caso de nuestro mazo de cartas, si sabemos que las cartas están en orden, podemos escribir una versión de `buscarCarta` que sea mucho más rápida. La mejor forma de escribir una búsqueda binaria es usando un método recursivo. Esto es porque este algoritmo es naturalmente recursivo.

El truco es escribir un método llamado `buscarBinario` que tome dos índices como parámetro, desde y hasta, indicando el segmento del arreglo en el cual se debe buscar (incluyendo desde y hasta).

1. Para buscar en el arreglo, elegir un índice entre desde y hasta (llamémoslo medio) y comparar la carta en esa posición con la carta que estás buscando.
2. Si la encontraste, parar.
3. Si la carta en medio es mayor que tu carta, buscar en el rango entre desde y mid-1.
4. Si la carta en medio es menor que tu carta, buscar en el rango entre mid+1 y hasta.

Los pasos 3 y 4 se ven sospechosamente como llamados recursivos. Aquí está cómo se vería esto traducido a código Java:

```
public static int buscarBinario (Carta[] mazo, Carta carta,
                                int desde, int hasta) {
    int medio = (hasta + desde) / 2;
    int comp = compararCarta (mazo[medio], carta);

    if (comp == 0) {
        return medio;
    } else if (comp > 0) {
        return buscarBinario (mazo, carta, desde, medio-1);
    } else {
        return buscarBinario (mazo, carta, medio+1, hasta);
    }
}
```

En lugar de llamar a compararCarta tres veces, la llamé una sola vez y almacené el resultado.

A pesar de que este código contiene el núcleo de la búsqueda binaria, aun le falta una pieza. Tal como está escrito, si la carta no está en el mazo, habrá una recursión infinita. Necesitamos una forma de detectar esta condición y manejarla apropiadamente (devolviendo -1).

La manera más sencilla de darse cuenta que la carta no está en el mazo es cuando *no* hay cartas en el mazo, lo cual es el caso en que hasta es menor que desde. Bueno, claro que aún hay cartas en el mazo, pero lo que quiero decir es que no hay más cartas en el segmento del mazo indicado por desde y hasta.

Con el agregado de esa línea, el método funciona correctamente:

```
public static int buscarBinario (Carta[] mazo, Carta carta,
                                int desde, int hasta) {
    System.out.println (desde + ", " + hasta);
```

```

    if (hasta < desde) return -1;

    int medio = (hasta + desde) / 2;
    int comp = compararCarta (mazo[medio], carta);

    if (comp == 0) {
        return medio;
    } else if (comp > 0) {
        return buscarBinario (mazo, carta, desde, medio-1);
    } else {
        return buscarBinario (mazo, carta, medio+1, hasta);
    }
}

```

Agregué una sentencia de impresión al principio para poder ver la secuencia de llamados recursivos y convencerme de que eventualmente llegará al caso base. Probé el siguiente código:

```

    Carta carta1 = new Carta (1, 11);
    System.out.println (buscarBinario (mazo, carta1, 0, 51));

```

Y obtuve la siguiente salida:

```

0, 51
0, 24
13, 24
19, 24
22, 24
23

```

Luego inventé una carta que no está en el mazo (el 15 de Diamantes), e intenté buscarla. Obtuve lo siguiente:

```

0, 51
0, 24
13, 24
13, 17
13, 14
13, 12
-1

```

Estos tests no prueban que el programa sea correcto. De hecho, ninguna cantidad de testing prueba que un programa es correcto. De todas maneras, inspeccionando algunos pocos casos y examinando el código, podés convencerte de que funciona.

La cantidad de llamados recursivos es bastante baja, típicamente 6 o 7. Esto indica que sólo tenemos que llamar a `compararCarta` 6 o 7 veces, comparado con las 52 veces si hiciéramos una búsqueda lineal. En general, la búsqueda binaria es mucho más rápida que la búsqueda lineal, y aún más para arreglos largos.

Dos errores comunes en los programas recursivos son olvidarse de incluir un caso base y escribir los llamados recursivos de manera tal que el caso base nunca se alcanza. Cualquiera de estos errores causará una recursión infinita, y en tal caso Java lanzará (eventualmente) una excepción de tipo `StackOverflowException`⁵.

11.9 Mazos y submazos

Examinando la interfaz de `buscarBinario`

```
public static int buscarBinario (Carta[] mazo, Carta carta,
                                int desde, int hasta) {
```

puede tener sentido pensar en tres de los parámetros, `mazo`, `desde` y `hasta`, como un único parámetro que especifica un **submazo**. Esta forma de pensar es bastante común, y yo a veces lo pienso como un **parámetro abstracto**. Con “abstracto”, me refiero a algo que no es literalmente parte del texto del programa, pero que describe el funcionamiento del programa en un nivel más alto.

Por ejemplo, cuando alguien llama a un método y le pasa un arreglo y los límites `desde` y `hasta`, no hay nada que evite que el método llamado acceda a partes del arreglo fuera de estos límites. Entonces no se está pasando literalmente un subconjunto del mazo; realmente se está mandando el mazo entero. Pero mientras el receptor juegue acorde a las reglas, tiene sentido pensarlo, abstractamente, como un submazo.

Hay otro ejemplo de este tipo de abstracción que tal vez hayas notado en la Sección 9.7, cuando hablé de una estructura de datos “vacía”. El motivo por el cual escribí “vacía” entre comillas fue para sugerir que esa palabra no era literalmente precisa. Todas las variables tienen un valor todo el tiempo. Cuando las creas, se les da valores por defecto. Por lo tanto no existe algo como un objeto vacío.

Pero si el programa garantiza que nunca se leerá el valor actual de una variable antes de que se le asigne otro, entonces el valor actual es irrelevante. Abstractamente, tiene sentido pensar a esa variable como “vacía”.

Esta forma de pensar, en la que un programa cobra significados más allá de lo que está literalmente codificado, es una parte muy importante

5. N.d.T: En inglés significa “desbordamiento de pila”.

del pensamiento de un científico de la computación. A veces la palabra “abstracto” se usa tan seguido y en contextos tan variados que se pierde su significado. Sin embargo, la abstracción es una de las ideas centrales de las ciencias de la computación (y de muchas otras ciencias).

Una definición mucho más genérica de “abstracción” es “El proceso de modelar un sistema complejo con una descripción simplificada para capturar el comportamiento relevante eliminando los detalles innecesarios”.

11.10 Glosario

codificar: Representar un conjunto de valores usando otro conjunto de valores, construyendo una relación (o *mapeo*) entre ellos.

igualdad superficial: Igualdad de referencias. Dos referencias que apuntan a un mismo objeto.

igualdad de contenido: Igualdad de valores. Dos referencias que apuntan a objetos que contienen los mismos valores.

parámetro abstracto: Conjunto de parámetros que actúan juntos como un único parámetro.

abstracción: Proceso de interpretar un programa (o cualquier otra cosa) en un nivel más alto de lo que está literalmente representado por su código.

11.11 Ejercicios

Ejercicio 11.2

Imaginá un juego de cartas en el cual el objetivo es obtener una mano con un puntaje total de 21. El puntaje de una mano es la suma de los puntajes de cada carta. El puntaje de una carta es el siguiente: los ases cuentan como 1, todas las figuras (J, Q y K) cuentan como diez; para el resto de las cartas el puntaje es el valor de las mismas. Ejemplo: la mano (As, 10, J, 3) tiene un puntaje total de $1 + 10 + 10 + 3 = 24$.

Escribí un método llamado `puntajeMano` que tome un arreglo de cartas como argumento y que sume (y devuelva) el puntaje total. Asumí que los valores de las cartas están codificados acorde al mapeo dado en la Sección 11.2, con los Ases codificados como 1.

Ejercicio 11.3

El método `imprimirCarta` de la Sección 11.3 toma un objeto de tipo `Carta` y devuelve un `String` representando la misma.

Escribí un método de clase para la clase `Carta` llamado `leerCarta` que tome una cadena y devuelva la carta correspondiente. Podés asumir que la cadena contiene el nombre de una carta en un formato válido, como si la misma se hubiese producido con `imprimirCarta`.

En otras palabras, la cadena contendrá un único espacio entre el valor y la palabra “de” y entre la palabra “de” y el palo. Si la cadena no contiene un nombre de carta válido, el método deberá devolver un objeto `null`.

El objetivo de este problema es repasar el concepto del análisis sintáctico (o *parsing*) e implementar un método que lea un determinado conjunto de cadenas.

Ejercicio 11.4

Escribí un método llamado `histoPalos` que tome un arreglo de cartas como parámetro y devuelva un histograma de los palos de la mano. La solución debe recorrer el arreglo una sola vez.

Ejercicio 11.5

Escribí un método llamado `esColor` que tome un arreglo de `Cartas` como parámetro y devuelva verdadero si la mano es un *color* y falso en caso contrario. Un *color* es una mano de poker que consta de cinco cartas del mismo palo.

Capítulo 12

Objetos como Arreglos

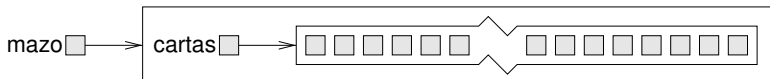
12.1 La clase `Mazo`

En el capítulo anterior, trabajamos con un arreglo de objetos, pero también se mencionó que es posible tener un objeto que contenga un arreglo como variable de instancia. En este capítulo vamos a crear una nueva clase de objetos, llamada `Mazo`, que contiene un arreglo de `Cartas` como variable de instancia.

La definición de la clase se ve así:

```
class Mazo {  
    Carta[] cartas;  
  
    public Mazo (int n) {  
        cartas = new Carta[n];  
    }  
}
```

El nombre de la variable de instancia es `cartas` para ayudar a distinguir el objeto `Mazo` del arreglo de `Cartas` que él contiene. Aquí vemos un diagrama de estado mostrando cómo se ve un objeto `Mazo` sin ninguna carta reservada:



Como es usual, el constructor inicializa la variable de instancia, pero en este caso usa el comando `new` para crear el arreglo de `cartas`. De todas

maneras, no crea ninguna carta para que vaya dentro de él. Para eso podemos escribir otro constructor que crea un mazo estándar de 52 cartas y lo llena con objetos de tipo Carta:

```
public Mazo () {
    cartas = new Carta[52];
    int indice = 0;
    for (int palo = 0; palo <= 3; palo++) {
        for (int valor = 1; valor <= 13; valor++) {
            cartas[indice] = new Carta (palo, valor);
            indice++;
        }
    }
}
```

Notar qué similar es este método a `construirMazo`, excepto que tuvimos que cambiar la sintaxis para hacerlo un constructor. Para llamarlo, usamos el comando `new`:

```
Mazo mazo = new Mazo();
```

Ahora que tenemos una clase `Mazo`, tiene sentido poner todos los métodos que involucran Mazos en la definición de la clase `Mazo`. Mirando a los métodos que hemos escrito hasta ahora, un candidato obvio es `imprimirMazo` (Sección 11.7). Aquí vemos cómo se ve, reescrito para trabajar con un objeto de tipo `Mazo`:

```
public static void imprimirMazo (Mazo mazo) {
    for (int i=0; i<mazo.cartas.length; i++) {
        Carta.imprimirCarta (mazo.cartas[i]);
    }
}
```

La cosa más obvia que tenemos que cambiar es el tipo del parámetro, de `Carta[]` a `Mazo`. El segundo cambio es que ahora no podemos escribir `mazo.length` para obtener la longitud de un arreglo, porque `mazo` es un objeto de tipo `Mazo` ahora, no un arreglo. Él contiene un arreglo, pero no es, en sí mismo, un arreglo. Por esa razón, tenemos que escribir `mazo.cartas.length` para obtener el arreglo del objeto de tipo `Mazo` y obtener la longitud del arreglo.

Por la misma razón, tenemos que usar `mazo.cartas[i]` para acceder un elemento del arreglo, en vez de usar simplemente `mazo[i]`. El último cambio es que la llamada a `imprimirCarta` tiene que decir explícitamente que `imprimirCarta` está definido en la clase `Carta`.

Para algunos de los otros métodos, no es obvio si deberían ser incluidos en la clase Carta o la clase Mazo. Por ejemplo, buscarCarta toma una Carta y un Mazo como argumentos; podrías razonablemente ponerlo en cualquiera de las dos clases. Como ejercicio, mové buscarCarta a la clase Mazo y reescribílo para que el primer parámetro sea un Mazo en vez de ser un arreglo de Cartas.

12.2 Mezclando

Para la mayoría de los juegos de cartas, es necesario que sea posible mezclar el mazo; es decir, poner las cartas en orden aleatorio. En la Sección 10.6 vimos cómo generar números aleatorios, pero no es obvio cómo usarlos para mezclar un mazo.

Una posibilidad es modelar la forma en que los humanos mezclan, que usualmente es dividiendo el mazo a la mitad, y después rearmando el mazo eligiendo alternadamente de cada mazo. Ya que los humanos usualmente no mezclan perfectamente, después de alrededor de 7 iteraciones el orden del mazo es bastante aleatorio. Pero un programa de computadora tendría la molesta propiedad de realizar una mezcla perfecta cada vez, lo cual no es realmente muy aleatorio. De hecho, después de 8 mezclas perfectas, encontrarás que el mazo está en exactamente el mismo orden en el que comenzaste. Para una discusión acerca de este hecho, entrá a <http://www.wiskit.com/marilyn/craig.html> o buscá en la web las palabras “perfect shuffle”.

Un mejor algoritmo de mezclado es recorrer el mazo una carta a la vez, y en cada iteración elegir dos cartas e intercambiarlas.

Aquí se muestra un esbozo de cómo funciona el algoritmo. Para esquematizar el programa, estoy usando una combinación de sentencias Java y palabras en castellano que a veces se llama **pseudocódigo**:

```
for (int i=0; i<mazo.length; i++) {  
    // elegir un numero aleatorio entre i y  
    // mazo.cartas.length intercambiar la  
    // i-esima carta con la elegida al azar  
}
```

Lo bueno de usar pseudocódigo es que a menudo deja en claro cuáles métodos van a ser necesarios. Acá, necesitamos algo como enteroAleatorio, que elija un entero aleatorio entre minimo y maximo, e intercambiarCartas que tome dos índices e intercambie las cartas en las posiciones indicadas.

Probablemente puedas imaginar cómo escribir enteroAleatorio mirando la Sección 10.6, aunque tenés que ser cuidadoso con respecto a la posibilidad de generar índices que estén fuera del rango.

También podrás imaginar intercambiarCartas vos mismo. El único problema está en decidir si intercambiar sólo las referencias a las cartas o el contenido de las cartas. ¿Es importante esa decisión? ¿Cuál forma es más rápida?

Queda como ejercicio la parte restante de la implementación de estos métodos.

12.3 Ordenamiento

Ahora que ya desordenamos el mazo, necesitamos una forma de ponerlo otra vez en orden. Irónicamente, hay un algoritmo para ordenarlo que es muy similar al algoritmo para mezclarlo. Este algoritmo es a veces llamado **ordenamiento por selección** porque trabaja recorriendo el arreglo repetidamente y seleccionando la carta más baja cada vez.

Durante la primera iteración, buscamos la carta más baja y la intercambiamos con la que está en la posición 0. Durante la i -ésima iteración, buscamos la carta más baja a la derecha de i y la intercambiamos con la i -ésima carta.

Este es el pseudocódigo del ordenamiento por selección:

```
for (int i=0; i<mazo.length; i++) {  
    // buscar la carta más baja a la derecha de i ó en i  
    // intercambiar esa carta con la i-ésima  
}
```

Otra vez, el pseudocódigo ayuda con el diseño de los **métodos auxiliares**. En este caso podemos usar intercambiarCartas otra vez, por lo que sólo necesitamos uno nuevo, llamado buscarCartaMasBaja, que toma un arreglo de cartas y un índice donde debería empezar a buscar.

Una vez más, la implementación queda como tarea para el lector.

12.4 Submazos

¿Cómo deberíamos representar una mano u otro subconjunto de un mazo entero? Una posibilidad es crear una nueva clase llamada Mano, que podría extender a Mazo. Otra posibilidad, la que voy a demostrar, es representar una mano con un objeto de tipo Mazo, que ocurre que tiene menos de 52 cartas.

Podríamos querer un método, submazo, que toma un Mazo y un rango de índices, y que devuelve el nuevo Mazo que contiene el subconjunto especificado de cartas:

```

public static Mazo submazo (Mazo mazo, int menor, int mayor) {
    Mazo sub = new Mazo (mayor - menor + 1);

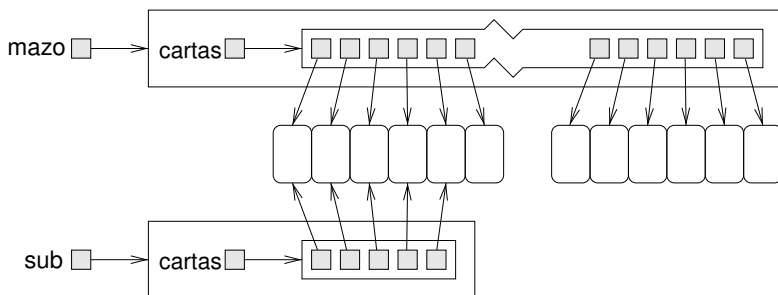
    for (int i = 0; i < sub.cartas.length; i++) {
        sub.cartas[i] = mazo.cartas[menor+i];
    }
    return sub;
}

```

La longitud del submazo es $\text{mayor} - \text{menor} + 1$ porque tanto la mayor como la menor carta están incluidas. Este tipo de cálculo puede ser confuso, y llevar a errores “por un elemento”. En general, es preferible tratar de evitarlos.

Como proveemos un argumento con el comando `new`, el constructor que es llamado será el primero, que sólo reserva el espacio del arreglo y no reserva ninguna carta. Dentro del `for`, el submazo es llenado con copias de las referencias al mazo.

El siguiente es un diagrama de estado de un submazo siendo creado con los parámetros `menor=3` y `mayor=7`. El resultado es una mano con 5 cartas que son compartidas con el mazo original; es decir, hacen aliasing.



En su momento, sugerí que el aliasing no es una buena idea, ya que cambios en un submazo serán reflejados en otros, lo cual es el comportamiento que esperarías de mazos y cartas reales. Pero si los objetos en cuestión son inmutables, entonces el aliasing es menos peligroso. En este caso, probablemente no haya razón alguna para cambiar el número o el palo de una carta. En vez de eso, crearemos cada carta y luego la trataremos como un objeto inmutable. Así, para las Cartas el aliasing es una decisión razonable.

12.5 Mezclando y repartiendo

En la Sección 12.2 escribimos un pseudocódigo para un algoritmo de mezclado. Asumiendo que tenemos un método llamado `mezclarMazo` que toma un mazo como argumento y lo mezcla, podemos crear y mezclar un mazo:

```
Mazo mazo = new Mazo ();  
mezclarMazo (mazo);
```

Después, para tratar con varias manos, podemos usar `submazo`:

```
Mazo mano1 = submazo (mazo, 0, 4);  
Mazo mano2 = submazo (mazo, 5, 9);  
Mazo baraja = submazo (mazo, 10, 51);
```

Este código pone las primeras 5 cartas en una mano, las siguientes 5 en el otro, y el resto en la baraja.

Cuando pensaste en la forma de repartir, ¿pensaste que deberíamos repartir una carta por vez a cada jugador en forma circular como es común en los juegos de cartas reales? Yo pensé en eso, pero después me di cuenta que eso no es necesario en un programa de computadora. La convención circular tiene como objetivo mitigar el mezclado imperfecto y hacer más difícil que quien reparte haga trampa. Ninguno de esos dos es un problema para una computadora.

Este ejemplo es útil para recordar uno de los peligros de las metáforas de diseño: a veces imponemos restricciones en las computadoras que son innecesarias, o esperamos capacidades que no están presentes, porque sin pensarlo estamos extendiendo una metáfora y pasando el punto en el que ya deja de tener sentido. Cuidado con las analogías engañosas.

12.6 Mergesort²

En la Sección 12.3, vimos un algoritmo de ordenamiento simple que resulta no ser muy eficiente. Para ordenar n ítems, hay que recorrer el arreglo n veces, y cada recorrida toma una cantidad de tiempo que es proporcional a n . El tiempo total, entonces, es proporcional a n^2 .

En esta sección vamos a esquematizar un algoritmo más eficiente llamado **mergesort**. Para ordenar n ítems, mergesort toma tiempo proporcional a $n \log n$. Esto puede no parecer impresionante, pero a medida que n se vuelve grande, la diferencia entre n^2 y $n \log n$ se vuelve enorme. Intentá haciendo la cuenta con algunos valores de n y vas a ver.

2. N.d.T.: Ordenamiento por mezcla.

La idea básica detrás de mergesort es esta: si tenés dos submazos, cada uno de los cuales ya ha sido ordenado, entonces es fácil (y rápido) unirlos en un único arreglo ordenado. Intentá esto con un mazo de cartas:

1. Formar dos submazos con cerca de 10 cartas cada uno y ordenarlos de mayor a menor para que la carta más baja quede en la cima. Dejar los dos mazos frente a uno.
2. Comparar la carta del tope de cada mazo y elegir la menor entre ambas. Tomarla y pasarla al mazo ordenado.
3. Repetir el paso dos hasta que uno de los mazos esté vacío. Después tomar las cartas restantes y agregarlas al mazo ordenado.

El resultado debería ser un único mazo ordenado. Así se ve en pseudo-código:

```
public static Mazo unir (Mazo mazo1, Mazo mazo2) {
    // crear un nuevo mazo suficientemente grande
    // para tener todas las cartas
    Mazo resultado = new Mazo (mazo1.cartas.length +
                               mazo2.cartas.length);

    // usar el índice i para llevar la cuenta de
    // dónde estamos en el primer mazo, y usar el
    // índice j para el segundo mazo
    int i = 0;
    int j = 0;

    // el índice k recorre el mazo resultante
    for (int k = 0; k<resultado.cartas.length; k++) {

        // si mazo1 está vacío, mazo2 gana;
        // si mazo2 está vacío, mazo1 gana;
        // si ninguno está vacío, comparar las dos cartas

        // agregar al ganador al nuevo mazo
    }
    return resultado;
}
```

La mejor manera de probar unir es construir y mezclar un mazo, usar un submazo para formar dos (pequeñas) manos, y después usar la rutina del capítulo anterior para ordenar las dos mitades. Entonces podemos pasar las dos mitades a unir para ver si funciona.

Si pudiste hacer que eso funcione, intentá implementar mergeSort:

```
public static Deck mergeSort (Mazo mazo) {  
    // encontrar el punto medio del mazo  
    // dividir el mazo en dos submazos  
    // ordenar los dos submazos usando ordenarMazo  
    // unir las dos mitades y devolver el resultado  
}
```

Después, si pudiste hacer que eso también funcione, empieza la diversión real! Lo mágico de mergesort es que es recursivo. En el punto en que se ordenan los submazos, ¿por qué debería llamarse a la versión vieja y lenta de sort? ¿Por qué no llamar al fantástico y nuevo mergesort que estamos escribiendo?

No sólo es una buena idea, es *necesario* para poder lograr la ventaja de eficiencia que habíamos prometido. Para poder hacerlo funcionar, sin embargo, hay que agregar un caso base para que no se ejecute recursivamente para siempre. Un caso base simple es un submazo con 0 o 1 cartas. Si mergesort recibe un submazo tan chico, puede devolverlo sin modificar, pues ya está ordenado.

La versión recursiva de mergesort debería verse así:

```
public static Deck mergeSort (Mazo mazo) {  
    // si el mazo es de 0 o 1 cartas, devolverlo  
  
    // encontrar el punto medio del mazo  
    // dividir el mazo en dos submazos  
    // ordenar los dos submazos usando mergeSort  
    // unir las dos mitades y devolver el resultado  
}
```

Como es usual, hay dos formas de pensar acerca de programas recursivos: se puede pensar a través de todo el flujo de ejecución, o se puede hacer un “salto de fe”. El siguiente ejemplo fue construido deliberadamente para alentarte a que hagas ese salto de fe.

Cuando estabas usando ordenarMazo para ordenar los submazos, ¿no te sentiste obligado a seguir el flujo de ejecución, verdad? Simplemente asumiste que el método ordenarMazo funcionaría porque ya lo habías depurado. Bueno, todo lo que hiciste para volver recursivo a mergeSort fue reemplazar un algoritmo de ordenamiento por otro. No hay razón para leer el programa de forma diferente.

Bueno, en realidad, vas a tener que pensar un poco para llegar hasta el caso base y asegurarte de que se alcanza eventualmente, pero más allá de eso, escribir una versión recursiva no debería ser un problema. ¡Buena suerte!

12.7 Glosario

pseudocódigo: Manera de diseñar programas escribiendo el código de forma esquematizada y mezclando Java y lenguaje natural.

método auxiliar: A menudo un pequeño método que no hace algo enormemente útil por sí mismo, pero que ayuda a otro método más útil.

12.8 Ejercicios

Ejercicio 12.1

Escribir una versión de `buscarBinario` que toma un submazo como argumento, en vez de un mazo y un rango de índices (ver Sección 11.8). ¿Cuál versión es más propensa a errores? ¿Cuál versión pensás que es más eficiente?

Ejercicio 12.2

En la versión anterior de la clase `Carta`, un mazo fue implementado como un arreglo de `Cartas`. Por ejemplo, cuando pasamos un “mazo” como parámetro, el tipo real del parámetro era `Carta[]`.

En este capítulo, desarrollamos una representación alternativa para un mazo, un objeto llamado `Mazo` que contiene un arreglo de cartas como variable de instancia. En este ejercicio, vas a implementar la nueva representación de un mazo.

- Agregar un segundo archivo, llamado `Mazo.java` al programa. Este archivo va a contener la definición de la clase `Mazo`.
- Escribir los constructores de `Mazo` como se muestra en la Sección 12.1.
- De los métodos que actualmente están en la clase `Carta`, decidir cuáles serían más apropiados como miembros de la nueva clase `Mazo`. Moverlos a ella y hacer todos los cambios necesarios para que el programa compile y ande otra vez.
- Mirar el programa e identificar todos los lugares donde un arreglo de `Cartas` está siendo usado para representar a un mazo. Modificar el programa en esos lugares para que use un objeto de tipo `Mazo`. Puede usarse la versión de `imprimirMazo` en la sección 12.1 como ejemplo.

Probablemente sea una buena idea hacer esta transformación de a un método a la vez, y probar el programa después de cada cambio. Por otro lado, si confiás en que sabés lo que estás haciendo, podés hacer la mayoría de los comandos con la herramienta de búsqueda y reemplazo del editor de texto.

Ejercicio 12.3

El objetivo de este ejercicio es implementar los algoritmos de mezclado y de ordenamiento de este capítulo.

- a. Escribir un método llamado `intercambiarCartas` que tome un mazo (arreglo de cartas) y dos índices, y que intercambie las cartas de esas dos posiciones.
 AYUDA: deberías intercambiar las referencias a las dos cartas, más que el contenido de los dos objetos. Esto no sólo es más rápido, sino que hace más fácil tratar con el caso de que las cartas tengan aliasing.
- b. Escribir un método llamado `mezclarMazo` que use el algoritmo de la Sección 12.2. Quizá quieras usar el método `enteroAleatorio` del Ejercicio 10.2.
- c. Escribir el método `buscarCartaMasBaja` que use el método `compararCarta` para encontrar la carta más baja en un rango del mazo (desde `indiceMenor` hasta `indiceMayor`, incluyendo a ambas).
- d. Escribir un método llamado `ordenarMazo` que ordene un mazo de cartas de menor a mayor.

Ejercicio 12.4

Para hacer la vida más difícil a los contadores de cartas, muchos casinos ahora usan máquinas de mezclado que pueden hacer mezclado incremental, lo cual significa que después de cada mano, las cartas usadas son devueltas al mazo y, en vez de mezclar nuevamente el mazo entero, las nuevas cartas se insertan en posiciones aleatorias.

Escribir un método llamado `mezclarIncrementalmente` que tome un mazo y una carta e inserta la carta en el mazo en una ubicación aleatoria. Este es un ejemplo de un **algoritmo incremental**.

Ejercicio 12.5

El objetivo de este ejercicio es escribir un programa que genere manos de póker al azar y las clasifique, para que podamos estimar la probabilidad de varias manos de póker. No te preocupes si no jugás al póker; te diremos todo lo que necesitas saber.

- a. Como precalentamiento, escribir un programa que use `mezclarMazo` para generar e imprimir cuatro manos de póker con cinco cartas cada una. ¿Obtuviste algo bueno? Aquí hay cuatro posibles manos de póker, en orden de valor creciente:

par: dos cartas con el mismo número

par doble: dos pares de cartas con el mismo número

pierna: tres cartas con el mismo número

escalera: cinco cartas con números consecutivos

color: cinco cartas del mismo palo

full: tres cartas con un número, dos con otro

póker: cuatro cartas con el mismo número

escalera real: cinco cartas con números consecutivos del mismo palo

- b. Escribir un método llamado `esColor` que tome un `Mazo` como parámetro y devuelva un booleano indicando si la mano contiene un `Color`.
- c. Escribir un método llamado `esPierna` que tome una mano y devuelve un booleano indicando si la mano contiene una `Pierna`.
- d. Escribir un ciclo que genere unas pocas miles de manos y chequee si contienen un `Color` o una `Pierna`. Estimar la probabilidad de que toque una de esas manos.
- e. Escribir métodos que testeen por las otras manos de póker. Algunos son más fáciles que otros. Quizá encuentres útil escribir algunos métodos auxiliares de uso general que puedan ser usados para más de un test.
- f. En algunos juegos de póker, a cada uno de los jugadores se le reparten siete cartas, y ellos forman su mano con las mejores cinco de las siete. Modificar el programa para generar manos de siete cartas y recomputar las probabilidades.

Ejercicio 12.6

Como desafío especial, pensar en algoritmos que chequeen varias manos de póker si hubiera comodines. Por ejemplo, si “los dos son comodín”, eso significa que si tenemos una carta con número 2, podemos usarla para representar cualquier carta del mazo.

Ejercicio 12.7

El objetivo de este ejercicio es implementar mergesort.

- a. Usando el pseudocódigo de la Sección 12.6, escribir un método llamado `unir`. Asegurarse de probarlo antes de usarlo como parte de `mergeSort`.
- b. Escribir una versión simple de `mergeSort`, la cual divide el mazo a la mitad, usa `ordenarMazo` para ordenar las dos mitades, y usa `unir` para crear un nuevo mazo completamente ordenado.
- c. Escribir una versión recursiva de `mergeSort`. Recordar que `ordenarMazo` es un modificador y que `mergeSort` es una función pura, lo que significa que son llamados de forma diferente:

```
ordenarMazo(mazo);           // modifica el mazo existente
mazo = mergeSort (mazo);    // reemplaza el mazo viejo con el nuevo
```


Capítulo 13

Programación Orientada a Objetos

13.1 Lenguajes de programación y estilos

Hay muchos lenguajes de programación en este mundo, y casi tantos como estilos de programación (algunas veces llamados paradigmas). Tres estilos que han aparecido en este libro son el procedural, el funcional, y el orientado a objetos. Si bien Java es usualmente pensado como un lenguaje orientado a objetos, en Java es posible escribir programas en cualquier estilo. El estilo que utilicé en este libro es más que nada procedural. Los programas existentes en Java y los paquetes preincorporados están escritos en una mezcla de los tres estilos, pero tienden a ser más orientados a objetos que los programas de este libro.

No es sencillo definir qué es la programación orientada a objetos, pero aquí tenemos algunas de sus características:

- Las definiciones de objetos (clases) usualmente se corresponden con objetos relevantes de la vida real. Por ejemplo, en el Capítulo 12.1, la creación de la clase *Mazo* fue un paso hacia la programación orientada a objetos.
- La mayoría de los métodos son métodos de objeto (aquellos que son llamados para un objeto) en vez de ser métodos de clase (aquellos que son llamados a secas). Hasta ahora, todos los métodos que hemos escrito son de clase. En este capítulo vamos a escribir algunos métodos de objeto.
- La característica de un lenguaje más asociada a la programación orientada a objetos es la **herencia**. Voy a hablar de herencia más adelante en este capítulo.

Recientemente, la programación orientada a objetos se volvió bastante popular, y hay personas que afirman que es superior a los otros estilos en varios aspectos. Espero que exponiéndote a una variedad de estilos te habré dado las herramientas necesarias para entender y evaluar esas afirmaciones.

13.2 Métodos de clase y de objeto

Hay dos tipos de métodos en Java, llamados **métodos de clase** y **métodos de objeto**. Hasta ahora, cada método que hemos escrito ha sido un método de clase. Los métodos de clase son identificados con la palabra clave `static` en la primera línea. Cualquier método que no tenga la palabra clave `static` es un método de objeto.

Pese a que no hemos escrito ningún método de objeto, hemos llamado a algunos. Cada vez que llamás a un método “sobre” un objeto, es un método de objeto. Por ejemplo, `charAt` y otros métodos que llamamos sobre objetos `String` son todos métodos de objeto.

Cualquier cosa que puede ser escrita como un método de clase, también puede ser escrita como un método de objeto, y vice versa. A veces es más natural usar uno u otro. Por razones que serán aclaradas pronto, los métodos de objeto son frecuentemente más cortos que sus respectivos métodos de clase.

13.3 El objeto actual

Cuando llamás a un método de objeto, ese objeto pasa a ser **el objeto actual**. Dentro del método, te podés referir a las variables de instancia del objeto actual por su nombre, sin necesidad de especificar el nombre del objeto.

También, te podés referir al objeto actual usando la palabra clave `this`¹. Ya hemos visto `this` usado en constructores. De hecho, podés pensar a los constructores como un tipo especial de métodos de objeto.

13.4 Números complejos

Como ejemplo corriente para el resto del capítulo vamos a considerar la definición de una clase para números complejos. Los números complejos son útiles para muchas ramas de la matemática y la ingeniería, y muchos cálculos se realizan utilizando aritmética compleja. Un número complejo es la suma de una parte real y una parte imaginaria, y usual-

1. N.d.T: Del inglés, “este”.

mente se escribe en la forma $x + yi$, donde x es la parte real, y es la parte imaginaria, e i representa la raíz cuadrada de -1 . Por lo tanto, $i \cdot i = -1$.

A continuación, una definición de clase para un nuevo tipo de objeto llamado `Complejo`:

```
class Complejo
{
    // variables de instancia
    double real, imag;

    // constructor
    public Complejo () {
        this.real = 0.0;  this.imag = 0.0;
    }

    // constructor
    public Complejo (double real, double imag) {
        this.real = real;
        this.imag = imag;
    }
}
```

No debería haber nada sorprendente aquí. Las variables de instancia son doubles que contienen las partes real e imaginaria. Los dos constructores son del tipo usual: uno no toma ningún parámetro y asigna valores por defecto a las variables de instancia, el otro toma parámetros idénticos a las variables de instancia. Como hemos visto antes, la palabra clave `this` se usa para hacer referencia al objeto que está siendo inicializado.

En el `main`, o cualquier otro lado en donde queramos crear objetos `Complejos`, tenemos la opción de crear el objeto y luego fijar las variables de instancia, o de hacer ambas cosas al mismo tiempo:

```
Complejo x = new Complejo ();
x.real = 1.0;
x.imag = 2.0;
Complejo y = new Complejo (3.0, 4.0);
```

13.5 Una función de números Complejos

Echemos un vistazo a algunas de las operaciones que podríamos querer realizar con 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 computa el valor absoluto. Escrita como un método de clase, se ve así:


```
// método de clase
public static double abs (Complejo c) {
    return Math.sqrt (c.real * c.real + c.imag * c.imag);
}
```

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 cual el método fue llamado). Por lo tanto, no recibe ningún parámetro:

```
// método de objeto
public double abs () {
    return Math.sqrt (real*real + imag*imag);
}
```

Quitó la palabra clave `static` para indicar que este es un método de objeto. Además, eliminé el parámetro innecesario. Dentro del método, puedo referirme a las variables de instancia `real` e `imag` por su nombre sin necesidad de especificar un objeto. Java sabe implícitamente que me estoy refiriendo a las variables de instancia del método actual. Si quisiera hacerlo explícito, podría haber usado la palabra clave `this`:

```
// object method
public double abs () {
    return Math.sqrt (this.real * this.real +
                      this.imag * this.imag);
}
```

Pero eso sería más largo y en realidad no más claro. Para llamar a este método, lo hacemos sobre un objeto, por ejemplo

```
Complejo y = new Complejo (3.0, 4.0);
double resultado = y.abs();
```

13.6 Otra función sobre números Complejos

Otra operación que podríamos querer realizar sobre números complejos es una suma. Se pueden sumar números complejos simplemente sumando las partes reales y las partes imaginarias respectivamente. Escrito como un método de clase, se ve como:

```
public static Complejo sumar (Complejo a, Complejo b) {
    return new Complejo (a.real + b.real, a.imag + b.imag);
}
```

Para llamar a este método, le pasaríamos ambos operandos como parámetros:

```
Complejo suma = sumar(x, y);
```

Escrito como un método de objeto, tomaría solamente un parámetro, el cual sumaría con el objeto actual:

```
public Complejo sumar (Complejo b) {  
    return new Complejo (real + b.real, imag + b.imag);  
}
```

De nuevo, podemos referirnos a las variables de instancia del objeto actual implícitamente, pero para referirnos a las variables de instancia de *b* debemos llamar a *b* implícitamente usando la notación de punto. Para llamar a este método, hay que llamarlo sobre alguno de los operandos y pasar el otro como parámetro.

```
Complejo suma = x.suma (y);
```

En estos ejemplos se puede ver que el objeto actual (*this*) puede tomar el lugar de uno de los parámetros. Por esta razón, el objeto actual es algunas veces llamado un **parámetro implícito**.

13.7 Un modificador

Como un ejemplo más, veremos a conjugar, que es un método modificador que transforma un número Complejo en su complejo conjugado. El complejo conjugado de $x + yi$ es $x - yi$.

Como método de clase, esto se ve como:

```
public static void conjugar (Complejo c) {  
    c.imag = -c.imag;  
}
```

Como método de objeto, se ve como

```
public void conjugar () {  
    imag = -imag;  
}
```

A esta altura deberías ir convenciéndote de que con 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 no estarás restringido a escribir un tipo de métodos o el otro. Deberías estar

igual de familiarizado con ambos, así podrás elegir cuál de los dos es el más apropiado para la operación que estás escribiendo.

Por ejemplo, creo que sumar debería ser escrito como un método de clase, porque es una operación simétrica entre dos operandos, y tiene sentido que ambos operandos aparezcan como parámetros. Para mí, se ve raro llamar al método en uno de los operandos y pasar el otro como parámetro.

Por otro lado, operaciones simples que se aplican a un solo objeto pueden ser escritas más concisamente como métodos de objeto (incluso aunque tomen algunos parámetros adicionales).

13.8 El método `toString`

Todos los objetos tienen un método llamado `toString`² que genera una representación del objeto en forma de cadena. Cuando imprimís un objeto usando `print` o `println`, Java llama al método `toString` del objeto. La versión por defecto de `toString` devuelve una cadena que contiene el tipo del objeto y un identificador único (ver Sección 9.6). Cuando se define un nuevo tipo de objeto, se puede **sobreescibir** el comportamiento por defecto proveyendo un método nuevo con el comportamiento deseado.

Aquí vemos cómo se podría ver `toString` para la clase `Complejo`

```
public String toString () {  
    return real + " + " + imag + "i";  
}
```

El tipo de retorno de `toString` es `String`, naturalmente, y no toma ningún parámetro. Podés llamar a `toString` de la manera usual:

```
Complejo x = new Complejo (1.0, 2.0);  
String s = x.toString ();
```

o podés llamarlo directamente a través de `println`:

```
System.out.println (x);
```

En este caso, la salida es `1.0 + 2.0i`.

Esta versión de `toString` no se ve bien si la parte imaginaria es negativa. Como ejercicio, escribí una versión mejor.

2. N.d.T: Del inglés, aCadena.

13.9 El método equals

Cuando usás el operador `==` para comparar dos objetos, lo que en realidad estás preguntando es, “¿Son estas dos cosas el mismo objeto?” Es decir, si ambos objetos referencian a la misma posición de memoria.

Para muchos tipos, esa no es la definición apropiada de la igualdad. Por ejemplo, dos números complejos son iguales si sus partes reales son iguales y sus partes imaginarias también. No necesariamente tienen que ser el mismo objeto.

Cuando definís un nuevo tipo de objeto, podés escribir tu propia definición de igualdad, mediante un método de objeto llamado `equals`. Para la clase `Complejo`, tendría la siguiente forma:

```
public boolean equals (Complejo b) {  
    return (real == b.real && imag == b.imag);  
}
```

Por convención, `equals` es siempre un método de objeto que devuelve boolean.

La documentación de `equals` que se encuentra en la clase `Object` provee algunas guías que deberías tener en cuenta cuando creás tu propia definición de igualdad:

El método `equals` implementa una relación de equivalencia:

- Es reflexiva: para cualquier valor `x`, `x.equals(x)` debe devolver `true`.
- Es simétrica: para cualquier par de valores `x` e `y`, `x.equals(y)` debe devolver `true` si y sólo si `y.equals(x)` devuelve `true`.
- Es transitiva: para cualesquiera valores `x`, `y`, `z`, si `x.equals(y)` devuelve `true` y `y.equals(z)` devuelve `true`, entonces `x.equals(z)` tiene que devolver `true`.
- Es consistente: para cualquier par de valores `x` e `y`, múltiples llamadas de `x.equals(y)` devuelven siempre `true` o siempre `false`.
- Para cualquier valor `x`, `x.equals(null)` debe devolver `false`.

La definición de `equals` que di satisface todas estas condiciones, excepto una. ¿Cuál? Como ejercicio, arreglalo.

13.10 Llamando a un método de objeto desde otro

Como podés esperar, es válido y común llamar a un método de objeto. Por ejemplo, para normalizar un número complejo, dividís ambas partes por el valor absoluto. Puede no ser obvio por qué esto es útil, pero lo es.

Escribamos un método normalizar como un método de objeto, y hagamos que sea un modificador.

```
public void normalizar () {  
    double d = this.abs();  
    real = real/d;  
    imag = imag/d;  
}
```

La primera línea calcula el valor absoluto del objeto actual llamando a `abs` sobre el objeto actual. En este caso nombré al objeto actual explícitamente, pero pude haberlo omitido. Si llamás a otro método de objeto, Java asume que lo estás llamando sobre el objeto actual.

Ejercicio 13.1

Reescribir `normalizar` como una función pura. Luego, reescribirlo como un método de clase.

13.11 Rarezas y errores

Si tenés tantos métodos de objeto como métodos de clase en la misma definición de clase, es fácil confundirse. Una forma común de organizar la definición de una clase es poner todos los constructores al principio, seguidos por todos los métodos de objeto, y luego todos los métodos de clase.

Podés tener un método de objeto y un método de clase con el mismo nombre, siempre y cuando no tengan el mismo número y tipo de parámetros. Como con otros tipos de sobrecarga, Java decide qué versión llamar mirando qué parámetros le estás pasando.

Ahora que sabemos lo que significa la palabra `static` probablemente dedujiste que `main` es un método de clase, lo que significa que no hay un “objeto actual” cuando se llama.

Dado que no hay un objeto actual en método de clase, es un error utilizar la palabra `this`. Si lo probás, obtendrías un mensaje de error como “Undefined variable: this”³. Además, no podés referirte a variables de instancia sin usar la notación de punto y especificando un nombre de objeto.

3. N.d.T.: “variable sin definir: this”.

Si lo probás, obtendrías “Can’t make a static reference to nonstatic variable...”⁴. Este no es uno de los mejores mensajes de error, ya que utiliza un vocabulario no estándar. Por ejemplo, con “variable no estática” se refiere a una “variable de instancia”. Pero una vez que sabés lo que significa, sabés lo que significa.

13.12 Herencia

La característica de los lenguajes más asociada a la programación orientada a objetos es la **herencia**. Herencia es la capacidad de definir una clase que es una versión modificada de una definida antes (incluyendo las clases preincorporadas).

La principal ventaja de la herencia es que podés agregar nuevos métodos o variables de instancia a clases existentes, sin modificar dichas clases. Esto es particularmente útil para las clases preincorporadas dado que no las podés modificar aunque quisieras.

La razón de que la herencia lleve ese nombre es que las nuevas clases “heredan” todas las variables y métodos de la clase preexistente. Extendiendo esta metáfora, la clase preexistente se llama la clase **padre**.

13.13 Rectángulos dibujables

Como ejemplo de herencia, vamos a tomar la clase `Rectangle` y vamos a hacerla “dibujable”. Es decir, vamos a crear una nueva clase llamada `RectangleDibujable` que va a tener todas las variables de instancia y métodos de `Rectangle`, más un método adicional llamado `dibujar` que tomará un objeto de tipo `Graphics` como parámetro y dibujará un rectángulo.

La definición de la clase tendría esta forma:

```
import java.awt.*;

class RectangleDibujable extends Rectangle {

    public void dibujar (Graphics g) {
        g.drawRect (x, y, width, height);
    }
}
```

Sí, eso es todo lo que hace falta para definir esta clase. La primera línea importa el paquete `java.awt`, que es donde están definidos `Rectangle` y `Graphics`.

4. N.d.T.: “No se puede hacer una referencia estática a una variable no estática...”.

La siguiente línea indica que la clase `RectangleDibujable` hereda de la clase `Rectangle`. La palabra reservada `extends` se usa para identificar la clase de la cual queremos heredar, llamada **clase padre**.

El resto es la definición del método `dibujar`, el cual referencia las variables de instancia `x`, `y`, `width` y `height`. Puede parecer extraño referenciar una variable de instancia que no aparece en la definición de la clase, pero recordá que se heredan de la clase padre. Para crear y dibujar un `RectangleDibujable`, podés usar lo siguiente:

```
public static void dibujar (Graphics g, int x, int y,
                           int width, int height) {
    RectangleDibujable rd = new RectangleDibujable ();
    rd.x = 10;           rd.y = 10;
    rd.width = 200;      rd.height = 200;
    rd.dibujar (g);
}
```

Los parámetros de `dibujar` son un objeto `Graphics` y los límites del área dibujable (no las coordenadas del rectángulo).

Podemos establecer las variables de instancia de `rd` y llamar métodos de la manera usual. Cuando llamamos a `dibujar`, Java llama al método que definimos en `RectangleDibujable`. Si hubiéramos llamado a `grow` o cualquier otro método de `Rectangle` sobre `rd`, Java hubiera sabido que debía utilizar el método definido en la clase padre.

13.14 La jerarquía de clases

En Java, todas las clases extienden alguna otra clase. La más básica se llama `Object`. No contiene variables de instancia, pero provee los métodos `equals` y `toString`, entre otros.

Todas las clases extienden a `Object`, incluyendo casi todas las clases que hemos escrito y muchas de las clases preincorporadas, como la clase `Rectangle`. Cualquier clase que no especifique explícitamente su clase padre, hereda automáticamente de `Object`.

Sin embargo, algunas cadenas de herencia son más largas. Por ejemplo, en el Apéndice D.6, la clase `Pizarra` extiende a `Frame`, que extiende a `Window`, que extiende a `Container`, que extiende a `Component`, que finalmente extiende a `Object`. Sin importar qué tan larga sea la cadena, `Object` está al final.

Todas las clases de Java se pueden organizar en un “árbol familiar” llamado jerarquía. Usualmente, `Object` aparece al tope del árbol, con todas las clases hijas por debajo. Si mirás la documentación de `Frame`, por ejemplo, vas a ver parte de la jerarquía de la que forma parte `Frame`.

13.15 Diseño orientado a objetos

La herencia es una característica poderosa. Algunos programas que serían complicados sin ella pueden ser escritos en forma simple y concisa gracias a ella. Además, la herencia puede facilitar la reutilización de código, dado que podés personalizar el comportamiento de clases preincorporadas sin necesidad de modificarlas.

Por el otro lado, la herencia puede hacer que los programas sean difíciles de leer, dado que a veces no está claro, cuando un método es llamado, qué definición se va a ejecutar. Por ejemplo, uno de los métodos que podés llamar en Pizarra es `getBounds`. ¿Podés encontrar documentación para `getBounds`? Resulta que `getBounds` está definida en el padre del padre del padre de Pizarra.

Además, muchas de las cosas que pueden hacerse usando la herencia pueden hacerse casi tan elegantemente (o aún más) sin ella.

13.16 Glosario

método de objeto: Método que se llama sobre un objeto, y opera sobre ese objeto, el cual es referenciado por la palabra reservada `this` y se denomina el “objeto actual” en el habla coloquial. Los métodos de objetos no tienen la palabra `static` en su declaración.

método de clase: Método con la palabra reservada `static`. Los métodos de clase no se llaman sobre objetos y no tienen un objeto actual.

objeto actual: Objeto sobre el cual se llama un método de objeto. Dentro del método, el objeto actual es referenciado por `this`.

this: Palabra reservada que referencia el objeto actual.

implícito: Algo que se deja sin decir o implicado. Dentro de un método de objeto, podés referenciar las variables de instancia implícitamente (sin mencionar el nombre del objeto).

explícito: Cualquier cosa que se describe por completo. Dentro de un método de clase, las referencias a las variables de instancia tienen que ser explícitas.

13.17 Ejercicios

Ejercicio 13.2

Transformar el siguiente método de clase en un método de objeto.


```
public static double abs (Complejo c) {  
    return Math.sqrt (c.real * c.real + c.imag * c.imag);  
}
```

Ejercicio 13.3

Transformar el siguiente método de objeto en un método de clase.

```
public boolean equals (Complejo b) {  
    return (real == b.real && imag == b.imag);  
}
```

Ejercicio 13.4

Este ejercicio es una continuación del Ejercicio 9.3. El propósito es practicar la sintaxis de los métodos de objeto y familiarizarnos con los mensajes de error relevantes.

- a. Transformar los métodos de la clase Racional de métodos de clase en métodos de objeto, y hacer los cambios apropiados en el main.
- b. Cometer algunos errores. Probá llamar métodos de clase como si fueran métodos de objeto y viceversa. Intentá obtener un sentido de lo que es válido y lo que no, y de los errores que obtenés cuando hacés algo mal.
- c. Pensar acerca de los pros y los contras de los métodos de clase y de objeto. ¿Cuál es (generalmente) más conciso? ¿Cuál es la forma más natural para expresar un cómputo (o quizás, más apropiadamente, qué tipo de cálculos pueden expresarse más naturalmente con cada estilo)?

Capítulo 14

Listas enlazadas

14.1 Referencias en objetos

En el capítulo pasado vimos que las variables de instancia de un objeto pueden ser arreglos y mencioné que pueden ser objetos también.

Una de las posibilidades más interesantes es que un objeto puede contener una referencia a otro objeto del mismo tipo. Existe una estructura de datos muy común, la **lista**, que aprovecha esta característica.

Las listas están compuestas de **nodos**, donde cada nodo contiene una referencia al próximo nodo en la lista. Además, cada nodo usualmente contiene una unidad de datos llamada **carga**. En nuestro primer ejemplo, la carga será simplemente un entero, pero más adelante vamos a escribir una lista **genérica** que puede contener objetos de cualquier tipo.

14.2 La clase Nodo

Como de costumbre, cuando escribimos una nueva clase, empezamos con las variables de instancia, uno o dos constructores y algún método que nos permita testear el mecanismo básico de crear y mostrar el nuevo tipo, como por ejemplo toString.

```
public class Nodo {  
    int carga;  
    Nodo prox;  
  
    public Nodo () {  
        carga = 0;  
        prox = null;  
    }  
}
```

```

public Nodo (int carga, Nodo prox) {
    this.carga = carga;
    this.prox = prox;
}

public String toString () {
    return carga + "";
}
}

```

Las declaraciones de las variables de instancia se deducen naturalmente de la especificación, y el resto es mecánico a partir de las variables de instancia. La expresión `carga + ""` es una manera, rara pero concisa, de convertir un entero a un `String`.

Para probar la implementación hecha hasta ahora, pondremos algo así en el `main`:

```

Nodo nodo = new Nodo (1, null);
System.out.println (nodo);

```

El resultado es simplemente

1

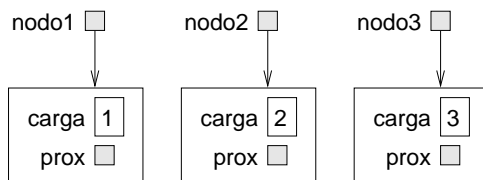
Para hacerlo interesante, ¡necesitamos una lista con más de un nodo!

```

Nodo nodo1 = new Nodo (1, null);
Nodo nodo2 = new Nodo (2, null);
Nodo nodo3 = new Nodo (3, null);

```

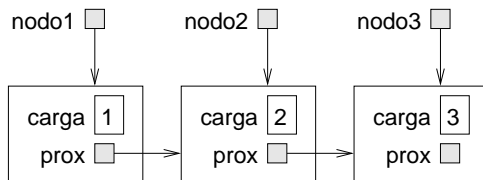
Este código crea tres nodos, pero no tenemos una lista aun ya que los nodos no están **enlazados**. El diagrama de estado se ve así:



Para enlazar los nodos, tenemos que hacer que el primer nodo referencie al segundo y el segundo al tercero.

```
nodo1.prox = nodo2;
nodo2.prox = nodo3;
nodo3.prox = null;
```

La referencia del tercer nodo es null, lo cual indica que es el final de la lista. Ahora el diagrama de estado se ve así:



Ahora ya sabemos cómo crear nodos y enlazarlos creando listas. Lo que puede no estar muy claro a esta altura es por qué hacer esto.

14.3 Listas como colecciones

El hecho que hace a las listas tan útiles es que son una manera de ensamblar muchos objetos en una sola entidad, llamada frecuentemente colección. En el ejemplo, el primer nodo de la lista sirve como una referencia a la lista entera.

Si queremos pasar la lista como parámetro a algún método, todo lo que necesitamos es pasar una referencia al primer nodo. Por ejemplo, el método `imprimirLista` toma un solo nodo como argumento. Empezando por el comienzo de la lista, imprime cada nodo hasta llegar al final (indicado por la referencia a null).

```
public static void imprimirLista (Nodo lista) {
    Nodo nodo = lista;

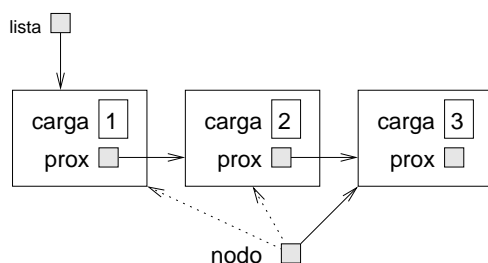
    while (nodo != null) {
        System.out.print (nodo);
        nodo = nodo.prox;
    }
    System.out.println ();
}
```

Para llamar a este método sólo tenemos que pasar una referencia al primer nodo:

```
imprimirLista (nodo1);
```

Dentro de `imprimirLista` tenemos una referencia al primer nodo de la lista, pero no tenemos ninguna variable que se refiera a los demás nodos. Tenemos que usar el valor de `prox` de cada nodo para llegar al próximo nodo.

Este diagrama de estado muestra el valor de `lista` y los valores que va tomando `nodo`:



Esta manera de moverse a través de una lista es llamada un **recorrido**, así como el patrón similar al moverse a través de los elementos de un arreglo. Es común usar un iterador como `nodo` para referenciar a cada nodo de la lista sucesivamente. La salida de este método es

123

Por convención, las listas se imprimen entre paréntesis con comas entre los elementos, como en `(1, 2, 3)`. A modo de ejercicio, modifiqué el método `imprimirLista` para que genere una salida con este formato.

Como otro ejercicio, reescribí `imprimirLista` usando un ciclo `for` en lugar de un ciclo `while`.

14.4 Listas y recursión

La recursión y las listas van juntas como las hamburguesas y las papas fritas. Por ejemplo, acá hay un algoritmo recursivo para imprimir el reverso de una lista:

1. Separar la lista en dos partes: el primer nodo (llamado la cabeza) y el resto (llamado la cola).
2. Imprimir el reverso de la cola.
3. Imprimir la cabeza.

Obviamente, el Paso 2, el llamado recursivo, asume que tenemos una forma de imprimir el reverso de una lista. Pero *si* asumimos que el llamado

recursivo funciona—el salto de fe—entonces podemos convencernos de que este algoritmo funciona.

Todo lo que necesitamos es el caso base, y una manera de probar que para cualquier lista, llegaremos eventualmente al caso base. Una elección natural para el caso base es una lista con un único elemento, aunque una mejor elección es la lista vacía, representada por null.

```
public static void imprimirInverso (Nodo lista) {  
    if (lista == null) return;  
  
    Nodo cabeza = lista;  
    Nodo cola = lista.prox;  
  
    imprimirInverso (cola);  
    System.out.print (cabeza);  
}
```

La primera línea se encarga del caso base sin hacer nada. Las siguientes dos líneas separan la lista en cabeza y cola. Las últimas dos líneas imprimen la lista.

Este método se llama exactamente igual que a imprimirLista:

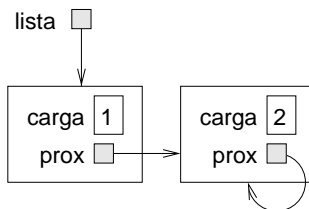
```
imprimirInverso(nodo1);
```

El resultado es el reverso de una lista.

¿Podemos demostrar que este método terminará siempre? En otras palabras, ¿llegará siempre al caso base? De hecho, la respuesta es que no. Existen algunas listas que harían que este método falle.

14.5 Listas infinitas

No hay manera de evitar que un nodo referencie a un nodo anterior de la lista, o incluso a sí mismo. Por ejemplo, esta figura muestra una lista de dos nodos, uno de los cuales se referencia a sí mismo.



Si llamamos imprimirLista en la lista, iterará por siempre.

Si llamamos `imprimirInverso` tendremos una resursión infinita. Este tipo de comportamiento hacen que las listas infinitas sean difíciles para trabajar.

Sin embargo, ocasionalmente son útiles. Por ejemplo, podríamos representar un número con una lista de dígitos, y usar una lista infinita para representar a los números periódicos.

De todas maneras, es un problema que no podamos demostrar que `imprimirLista` e `imprimirInverso` terminen. Lo mejor que podemos decir es la hipotética afirmación, “Si la lista no contiene ciclos, entonces estos métodos terminan”. A este tipo de afirmaciones se las llama **pre-condición**. Impone una condición sobre uno de los parámetros y describe el comportamiento del método si la condición se cumple. Veremos más ejemplos pronto.

14.6 El teorema fundamental de la ambigüedad

Hay un fragmento de `imprimirInverso` que puede haber sonado un poco raro:

```
Nodo cabeza = lista;  
Nodo cola = lista.prox;
```

Después de la primera asignación, `cabeza` y `lista` tienen el mismo tipo y el mismo valor. Entonces, ¿para qué creé una nueva variable?

El motivo es que estas dos variables juegan un rol diferente. Pensamos a `cabeza` como una referencia a un simple nodo, y pensamos a `lista` como una referencia al primer nodo de la lista. Estos “roles” 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 lo pensamos como una lista. Así que, aunque `cabeza` y `cola` tienen el mismo tipo, tienen distintos roles.

Esta ambigüedad es útil, pero puede hacer que los programas que tienen listas sean difíciles de leer. Yo suelo usar nombres de variables como `nodo` y `lista` para describir cómo pretendo usar una variable, y a veces creo variables adicionales para quitar esta ambigüedad.

Hubiera podido escribir `imprimirInverso` sin usar `cabeza` y `cola`, pero yo creo que lo hace más difícil de entender:

```
public static void imprimirInverso (Nodo lista) {  
    if (lista == null) return;  
  
    imprimirInverso (lista.prox);  
    System.out.print (lista);  
}
```

Examinando los dos llamados a función, tenemos que acordarnos que `imprimirInverso` trata a su argumento como una lista mientras que `print` trata a su argumento como un objeto simple.

Siempre es bueno tener en mente el **teorema fundamental de la ambigüedad**:

Una variable que referencia a un nodo puede tratar al nodo como un simple objeto o como el primer nodo de una lista.

14.7 Métodos de instancia para nodos

Probablemente, te debes estar preguntando por qué `imprimirLista` e `imprimirInverso` son métodos de clase. Yo afirmé que todo lo que pueda hacerse con métodos de clase también puede hacerse con métodos de instancia; es sólo cuestión de ver de qué forma queda más prolijo.

En este caso hay una razón legítima para elegir métodos de clase. Es válido mandar `null` como argumento de un método de clase, pero no es válido llamar un método de instancia sobre un objeto nulo.

```
Nodo nodo = null;
imprimirLista (nodo);      // valido
nodo.imprimirLista ();     // NullPointerException
```

Esta limitación dificulta escribir código para manipular listas en forma prolija y orientada a objetos. De todas maneras, un poco más tarde veremos una forma de solucionar este problema.

14.8 Modificando listas

Obviamente una forma de modificar una lista es cambiar la carga de uno de los nodos, pero las operaciones más interesantes son las que agregan, eliminan o reordenan los nodos.

Como ejemplo, vamos a escribir un método que elimina el segundo nodo en la lista y devuelve una referencia al nodo eliminado.

```
public static Nodo eliminarSegundo (Nodo lista) {
    Nodo primero = lista;
    Nodo segundo = lista.prox;

    // hace que el primer nodo reference al tercero
    primero.prox = segundo.prox;
```



```

    // separa el segundo nodo del resto de la lista
    segundo.prox = null;
    return segundo;
}

```

Otra vez, uso variables temporales para que el código quede más claro. Así es cómo se usaría este método:

```

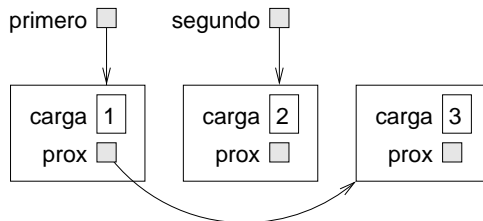
imprimirLista (nodo1);
Nodo eliminado = eliminarSegundo (nodo1);
imprimirLista (eliminado);
imprimirLista (nodo1);

```

La salida sería

(1, 2, 3)	la lista original
(2)	el nodo eliminado
(1, 3)	la lista modificada

Acá está el diagrama de estado mostrando el efecto de esta operación.



¿Qué pasaría si llamamos a este método y le pasamos una lista con un solo elemento (un **singleton**)? ¿Qué pasa si le pasamos una lista vacía como parámetro? ¿Tiene alguna precondition este método?

14.9 Adaptadores y auxiliares²

Para algunas de las operaciones sobre listas, nos es útil dividir el trabajo en dos métodos. Por ejemplo, para imprimir el inverso de una lista con el formato convencional, (3, 2, 1) podemos usar `imprimirInverso` para imprimir 3, 2, pero necesitamos un método separado para imprimir el paréntesis y el primer nodo. Lo llamaremos `imprimirInversoElegante`.

2. N.d.T.: Es más común ver estas dos palabras escritas en inglés: *wrappers* y *helpers*, respectivamente.

```

public static void imprimirInversoElegante (Nodo lista) {
    System.out.print ("");

    if (lista != null) {
        Nodo cabeza = lista;
        Nodo cola = lista.prox;
        imprimirInverso (cola);
        System.out.print (cabeza);
    }
    System.out.println ("");
}

```

De nuevo, es una buena idea verificar los métodos como este para ver si funcionan con casos especiales tales como una lista vacía o un singleton.

En cualquier lugar del programa, cuando usemos este método, llamaremos a `imprimirInversoElegante` y éste llamará a `imprimirInverso` por nosotros. En ese sentido, `imprimirInversoElegante` actúa como un adaptador o **wrapper** y usa `imprimirInverso` como un auxiliar o **helper**.

14.10 La clase `ListaInt`

Hay una serie de problemas sutiles con la manera en la que venimos implementando las listas. Dando vuelta un poco las cosas, primero voy a proponer una implementación alternativa y después voy a explicarte qué problemas se resuelven con ella.

Vamos a crear una nueva clase llamada `ListaInt`. Sus variables de instancia son un entero que contiene la longitud de la lista y una referencia al primer nodo en la lista. Los objetos `ListaInt` actúan como manipuladores de listas de objetos de tipo `Nodo`.

```

public class ListInt {
    int longitud;
    Node cabeza;

    public ListInt () {
        longitud = 0;
        cabeza = null;
    }
}

```

Una de las cosas lindas que tiene la clase `Listaint` es que nos da el lugar indicado para escribir funciones *wrapper* como `imprimirInversoElegante`, la cual podemos escribir como un método de objeto en la clase `ListaInt`.

```

public void imprimirInverso () {
    System.out.print ("(");

    if (cabeza != null) {
        Nodo cola = cabeza.prox;
        Nodo.imprimirInverso (cola);
        System.out.print (cabeza);
    }
    System.out.println (")");
}

```

Sólo para confundir, renombré `imprimirInversoElegante`. Ahora hay dos métodos llamados `imprimirInverso`: uno en la clase `Nodo` (el auxiliar) y uno en la clase `ListaInt` (el adaptador). Para que el adaptador pueda llamar al auxiliar, el adaptador tiene que identificar explícitamente la clase (`Nodo.imprimirInverso`).

Así, uno de los beneficios de la clase `ListaInt` es que provee un buen lugar en donde escribir funciones de tipo adaptador. Otro beneficio es que permite fácilmente agregar o sacar el primer elemento de una lista. Por ejemplo, `agregarAdelante` es un método de objeto para `ListaInt`; este método toma un `int` como argumento y lo agrega al principio de la lista.

```

public void agregarAdelante (int i) {
    Nodo nodo = new Nodo (i, cabeza);
    cabeza = nodo;
    longitud++;
}

```

Como siempre, para verificar este tipo de códigos sirve mucho pensar en los casos especiales. Por ejemplo, ¿qué pasa si la lista estaba vacía?

14.11 Invariantes

Algunas listas están “bien formadas”; otras no. Por ejemplo, si una lista tiene un ciclo, eso haría que varios de nuestros métodos dejaran de funcionar como deben, así que deberíamos pedir que las listas no contengan ciclos. Otro requerimiento es que el valor de `longitud` en un objeto `ListaInt` debe ser igual al número actual de nodos de la lista.

A este tipo de requerimientos se los llama **invariantes** ya que, idealmente, deben cumplirse para todo objeto todo el tiempo. Especificar invariantes para los objetos es una práctica de programación muy útil ya que facilita demostrar la corrección del código, verificar la integridad de las estructuras de datos y detectar errores.

Algo que es un poco confuso sobre los invariantes es que a veces éstos dejan de cumplirse. Por ejemplo, en la mitad de de agregarPrimero, después de haber agregado el nodo, pero antes de incrementar longitud, el invariante no se cumple. Este tipo de violación al invariante es aceptable; de hecho, usualmente es imposible modificar un objeto sin violar un invariante al menos por un momento. Normalmente el requerimiento es que todo método que viole un invariable debe restablecerlo antes de terminar.

Si en algún lado hay algún fragmento de código significativo en el cual el invariante es violado, sería importante dejarlo claro con comentarios, para que no se llame a ninguna operación que dependa del invariante.

14.12 Glosario

lista: Estructura de datos que implementa una colección usando una secuencia de nodos enlazados.

nodo: Elemento de una lista, usualmente implementado como un objeto que contiene una referencia a otro objeto del mismo tipo.

carga: Dato contenido en un nodo.

enlace: Referencia a un objeto contenida en otro objeto.

estructura de datos genérica: Tipo de estructura de datos que puede contener datos de cualquier tipo.

precondición: Afirmación que debe ser cierta para asegurar que un método funcione correctamente.

invariante: Afirmación sobre un objeto que debe ser cierta en todo momento (excepto tal vez mientras el objeto está siendo modificado).

método adaptador o wrapper: Método que actúa de intermediario entre el que llama al método y el método auxiliar, usualmente dando una interfaz un poco más prolija que la del método ayudante.

14.13 Ejercicios

Ejercicio 14.1

Empezá descargando el archivo `ListaInt.java` desde

<https://sourceforge.net/projects/thinkcsjava2esp/files/ListaInt.java>

Este archivo tiene las definiciones de `ListaInt` y `Nodo` de este capítulo, junto con código que muestra y testea algunos de los métodos. Compilá y ejecutá el programa. La salida debería ser algo así:

(1, 2, 3)
(3, 2, 1)

Los siguientes ejercicios te piden que escribas métodos de objeto adicionales en la clase `ListaInt`, pero probablemente te convenga escribir algunos métodos ayudantes en la clase `Nodo` también.

Después de escribir cada método, agregá código al `main` para probarlo. Asegurate de probar casos especiales como listas vacías y singletons.

Para cada método, identificá las precondiciones que sean necesarias para que el método funcione correctamente y agregá comentarios para documentarlas. Tus comentarios deberían indicar también si el método es un constructor, una función o un modificador.

- a. Escribí un método llamado `quitarPrimero` que elimina el primer nodo de una lista y devuelve su carga.
- b. Escribí un método llamado `asignar` que tome un índice, `i`, y un ítem de carga, y que reemplace la carga del `i`-ésimo nodo con la carga dada.
- c. Escribir un método llamado `agregar` que tome un índice, `i`, y un ítem de carga, y agregue un nuevo nodo que contenga la carga dada en la `i`-ésima posición.
- d. Escribí un método llamado `agregarAtras` que tome un ítem de carga y lo agregué al final de la lista.
- e. Escribí un método llamado `invertir` que modifique una `ListaInt`, invirtiendo el orden de los nodos.
- f. Escribí un método llamado `anexar` que tome una `ListaInt` como parámetro y anexe una copia de los nodos de la lista parámetro a la lista actual. Deberías poder reutilizar algunos de los métodos que ya escribiste.
- g. Escribí un método llamado `verificarLongitud` que devuelva `true` si el campo `longitud` es igual a la cantidad de nodos en la lista, y `false` en caso contrario. Este método no debería provocar una excepción en ningún caso, y debe terminar incluso si la lista contiene un ciclo.

Ejercicio 14.2

Una manera de representar números muy grandes es usando una lista de dígitos, usualmente almacenados en orden inverso. Por ejemplo, el número 123 podría ser representado con la lista (3, 2, 1).

Escribí un método que compare dos números representados con listas de tipo `ListaInt` y que devuelva 1 si el primero es mayor, -1 si el segundo es mayor, y 0 si son iguales.

Capítulo 15

Pilas

15.1 Tipos de datos abstractos

Los tipos de datos que hemos mirado hasta ahora son todos concretos, en el sentido de que hemos especificado completamente cómo se implementan. Por ejemplo, la clase `Carta` representa una carta usando dos enteros. Como discutimos en su momento, esa no es la única forma de representar una carta: hay muchas implementaciones alternativas.

Un **tipo de dato abstracto**, o TAD, especifica un conjunto de operaciones (o métodos) y la semántica de las operaciones (que es lo que hacen) pero no especifica la implementación de las mismas. Eso es lo que las hace abstractas.

¿Por qué es útil?

- Simplifica la tarea de especificar un algoritmo si podemos denotar las operaciones que necesitamos sin tener que pensar al mismo tiempo cómo esas operaciones se realizan.
- Ya que hay usualmente muchas maneras distintas de implementar un TAD, podría ser útil escribir un algoritmo que pueda ser usado con cualquiera de las posibles implementaciones.
- Existen TADs comúnmente reconocidos, como el TAD `Pila` en este capítulo, que a menudo son implementados en bibliotecas estándar de manera que sean escritos una sola vez y luego sean reutilizados por muchos programadores.
- Las operaciones en TADs proveen un lenguaje de alto nivel para especificar y hablar de algoritmos.

Cuando hablamos de TADs, a menudo distinguimos el código que usa al TAD, llamado el **código cliente**, del código que implementa al TAD, llamado el **código proveedor**, ya que provee un conjunto de servicios estándar.

15.2 El TAD Pila

En este capítulo vamos a analizar un TAD muy común, el de las pilas. Una pila es una colección, lo cual significa que es una estructura de datos que contiene múltiples elementos. Otras colecciones que hemos visto son los arreglos y las listas.

Como dijimos antes, un TAD se define por un conjunto de operaciones. Las pilas pueden realizar las siguientes operaciones:

constructor: Crea una nueva pila vacía.

apilar: Agregar un nuevo elemento al final de la pila.

desapilar: Quitar y devolver un ítem de la pila. El ítem devuelto es siempre el último que fue agregado.

estaVacía: Responder si la pila está vacía.

Una pila es a veces llamada una estructura LIFO, del inglés “last in, first out,” es decir “último en entrar, primero en salir”, porque el último ítem agregado es el primero en ser removido.

15.3 El objeto Stack de Java

Java provee un tipo de objeto preincorporado llamado Stack ¹ que implementa el TAD Pila. Deberías hacer un esfuerzo por mantener esas dos cosas—el TAD y la implementación Java—en su lugar. Antes de usar la clase Stack, tenemos que importarla desde `java.util`.

Las operaciones del TAD en la clase Stack de Java tienen los siguientes nombres:

apilar: `push`

desapilar: `pop`

estaVacía: `isEmpty`

Entonces la sintaxis para construir una nueva Stack es

1. N.d.T.: Stack: Pila en inglés.

```
Stack pila = new Stack ();
```

Inicialmente la pila está vacía, como podemos confirmar con el método `isEmpty`, que devuelve un boolean:

```
System.out.println (pila.isEmpty ());
```

Una pila es una estructura de datos genérica, lo cual significa que podemos agregar cualquier tipo de ítems a ella. En la implementación de Java, sin embargo, sólo podemos agregar objetos y no valores de tipos nativos.

Para nuestro primer ejemplo, vamos a usar objetos de tipo `Node`, como definimos en el capítulo anterior. Empecemos creando e imprimiendo una lista corta.

```
ListaInt lista = new ListaInt();  
lista.agregarAdelante (3);  
lista.agregarAdelante (2);  
lista.agregarAdelante (1);  
lista.imprimir ();
```

La salida es (1, 2, 3). Para poner un objeto de tipo `Nodo` en la pila, usamos el método `push`:

```
pila.push (lista.cabeza);
```

El siguiente ciclo recorre la lista y apila todos los nodos en la pila:

```
for (Nodo nodo = lista.cabeza; nodo != null; nodo = nodo.prox) {  
    pila.push (nodo);  
}
```

Podemos remover un elemento de la pila con el método `pop`.

```
Object obj = pila.pop ();
```

El tipo de retorno de `pop` es `Object`! Esto es porque la implementación de la pila no sabe exactamente de qué tipo son los objetos que contiene. Cuando apilamos los objetos de tipo `Nodo`, son convertidos automáticamente en `Objects`. Cuando los obtenemos de nuevo desde el stack tenemos que castearlos de nuevo a `Nodo`.

```
Nodo nodo = (Nodo) obj;  
System.out.println (nodo);
```


Desafortunadamente el manejo cae en el programador, que debe llevar la cuenta de los objetos en la pila y castearlos al tipo original cuando son removidos. Si se trata de castear un objeto a un tipo incorrecto se obtiene una `ClassCastException`.

El siguiente ciclo es la forma usual de recorrer la pila, desapilando todos los elementos y frenando cuando queda vacía:

```
while (!pila.isEmpty ()) {  
    Node nodo = (Node) pila.pop ();  
    System.out.print (nodo + " ");  
}
```

La salida es 3 2 1. En otras palabras, acabamos de usar una pila para imprimir los elementos de una lista de atrás para adelante! Por supuesto, este no es el formato estándar para imprimir una lista, pero usando una pila fue realmente fácil de hacer.

Podrías comparar este código con las implementaciones del capítulo anterior de `imprimirInverso`. Hay un paralelismo natural entre la versión recursiva de `imprimirInverso` y el algoritmo que usa una pila aquí. La diferencia es que `imprimirInverso` usa el stack de tiempo de ejecución para llevar la cuenta de los nodos mientras recorre la lista, y después los imprime a la vuelta de la recursión. El algoritmo que usa la pila hace lo mismo, pero usando un objeto de tipo `Stack` en vez de la pila de tiempo de ejecución.

15.4 Clases adaptadoras

Para cada tipo primitivo en Java, hay un tipo de objeto preincorporado llamado una **clase adaptadora**². Por ejemplo, la clase adaptadora de `int` se llama `Integer`; la de `double` se llama `Double`.

Las clases adaptadoras son útiles por varias razones:

- Es posible instanciar una clase adaptadora y crear objetos que contengan valores primitivos. En otras palabras, se puede adaptar un valor primitivo a un objeto, lo cual es útil si se quiere llamar a un método que requiere algo de tipo objeto.
- Cada clase adaptadora contiene valores especiales (como el mínimo y el máximo valor para ese tipo), y métodos que son útiles para convertir entre tipos.

2. N.d.T.: Del inglés `Wrapper class`.

15.5 Creando adaptadores

La forma más directa de crear un adaptador es usar su constructor:

```
Integer i = new Integer (17);  
Double d = new Double (3.14159);  
Character c = new Character ('b');
```

Técnicamente, `String` no es una clase adaptadora, porque no hay un correspondiente tipo primitivo, pero la sintaxis para crear un objeto de tipo `String` es la misma:

```
String s = new String ("alberto");
```

Por otro lado, nadie usa el constructor de objetos de tipo `String`, ya que se puede obtener el mismo efecto con un simple valor `String`:

```
String s = "alberto";
```

15.6 Creando más adaptadores

Algunas de las clases adaptadoras tienen un segundo constructor que toma un `String` como argumento y trata de convertirlo al tipo apropiado. Por ejemplo:

```
Integer i = new Integer ("17");  
Double d = new Double ("3.14159");
```

Este tipo de proceso de conversión no es muy robusto. Por ejemplo, si las `Strings` no están en el formato adecuado, la conversión causará una `NumberFormatException`. Cualquier carácter no numérico en la `String`, incluyendo espacio, causará que la conversión falle.

```
Integer i = new Integer ("17.1");           // ¡¡MAL!!  
Double d = new Double ("3.1459 ");         // ¡¡MAL!!
```

Es usualmente una buena idea validar el formato de la `String` antes de tratar de convertirla.

15.7 Sacando los valores afuera

Java sabe cómo imprimir adaptadores, por lo que la forma más simple de extraer su valor es simplemente imprimir el objeto:

```
Integer i = new Integer (17);
Double d = new Double (3.14159);
System.out.println (i);
System.out.println (d);
```

Alternativamente, se puede usar el método `toString` para convertir el contenido del adaptador en una `String`.

```
String istring = i.toString();
String dstring = d.toString();
```

Finalmente, si lo que se quiere es extraer el valor primitivo del objeto, hay un método correspondiente en cada adaptador que hace el trabajo:

```
int iprim = i.intValue ();
double dprim = d.doubleValue ();
```

También hay métodos para convertir adaptadores en valores primitivos de diferentes tipos. Deberías chequear la documentación de cada clase adaptadora para ver qué es lo que hay disponible.

15.8 Métodos útiles en las clases adaptadoras

Como mencionamos antes, las clases adaptadoras contienen métodos útiles que pertenecen a cada tipo. Por ejemplo, la clase `Character` contiene montones de métodos para convertir caracteres a mayúsculas y minúsculas, y para chequear si un carácter es un número, una letra o un símbolo.

La clase `String` también contiene métodos para convertir a mayúsculas y minúsculas. No olvides, sin embargo, que son funciones, no modificadores (ver Sección 7.9).

Para dar otro ejemplo, la clase `Integer` contiene métodos para interpretar e imprimir enteros en diferentes bases. Si tenemos una `String` que contiene un número en base 6, podemos convertirlo a base 10 usando `parseInt`.

```
String base6 = "12345";
int base10 = Integer.parseInt (base6, 6);
System.out.println (base10);
```

Ya que `parseInt` es un método de clase, se llama nombrando a la clase y al método en notación punto.

Usar base 6 puede no ser tan útil, pero la hexadecimal (base 16) y la octal (base 8) son comunes en cosas relacionadas con la ciencia de la computación.

15.9 Notación polaca

En la mayoría de los lenguajes de programación, las expresiones matemáticas se escriben con el operador entre los operandos, como en $1+2$. A este formato se lo denomina **infijo**. Un formato alternativo usado por algunas calculadoras se llama **sufijo** también conocido como **notación polaca**. En notación polaca, el operador sigue a los operandos, como en $1\ 2+$.

La razón por la cual la notación polaca es a veces útil es que hay una forma natural de evaluar una expresión en notación polaca usando una pila.

- Comenzando por el principio de la expresión, tomar un término (operador u operando) a la vez.
 - Si el término es un operando, apilarlo en la pila.
 - Si el término es un operador, desapilar dos operandos de la pila, realizar la operación en ellos y apilar el resultado de nuevo en la pila.
- Al llegar al final de la expresión, debería quedar exactamente un operando guardado en la pila. Ese operando es el resultado.

Como ejercicio, apliquemos este algoritmo a la expresión $1\ 2\ +\ 3\ *$.

Este ejemplo demuestra otra de las ventajas de la notación polaca: no hay necesidad de usar paréntesis para controlar el orden de las operaciones. Para obtener el mismo resultado con notación infija, deberíamos escribir $(1 + 2) * 3$. Como ejercicio, ¿podrías escribir en notación polaca una expresión que sea equivalente a $1 + 2 * 3$?

15.10 Parseo

Con el objetivo de implementar el algoritmo de la sección anterior, necesitamos ser capaces de recorrer una cadena y desarmarla en operandos y operadores. Este proceso es un ejemplo de **parseo**, y los resultados—los pedacitos individuales de la cadena—se denominan **tokens** o componentes léxicos.

Java provee una clase preincorporada llamada `StringTokenizer` que parsea cadenas y las convierte en tokens. Para usarla, antes hay que importarla desde `java.util`.

En su forma más simple, el `StringTokenizer` usa espacios para marcar la separación entre tokens. Un carácter que marca un límite es denominado un **delimitador**.

Podemos crear un `StringTokenizer` de la forma usual, pasando como argumento la cadena que queremos parsear.

```
StringTokenizer st = new StringTokenizer("Aquí hay cuatro tokens.");
```

El siguiente ciclo muestra la manera estándar para extraer los tokens del `StringTokenizer`³.

```
while (st.hasMoreTokens ()) {  
    System.out.println (st.nextToken());  
}
```

La salida es

```
Aquí  
hay  
cuatro  
tokens.
```

Para parsear expresiones, tenemos la opción de especificar caracteres especiales que serán usados como delimitadores:

```
StringTokenizer st = new StringTokenizer("11 22+33*", " +-*/");
```

El segundo argumento es una `String` que contiene todos los caracteres que van a ser usados como delimitadores. Esta vez la salida es:

```
11  
22  
33
```

Esto funciona para extraer todos los operandos pero perdimos los operadores. Afortunadamente, hay una opción más para los `StringTokenizers`.

```
StringTokenizer st = new StringTokenizer("11 22+33*", " +-*/", true);
```

El tercer argumento indica: "Sí, queremos tratar los delimitadores como tokens también". Ahora la salida es:

```
11  
  
22  
+  
33  
*
```

Este es justamente el flujo de tokens que queríamos para evaluar esta expresión.

3. N.d.T.: `hasMoreTokens` equivale a `tieneMásTokens`, y `nextToken` a `siguienteToken`.

15.11 Implementando TADs

Uno de los objetivos fundamentales de un TAD es el de separar los intereses del proveedor, que escribe el código que implementa el TAD, del cliente, que es quien lo usa. El proveedor sólo tiene que preocuparse de que la implementación sea correcta—de acuerdo con la especificación del TAD— y no de cómo será usado.

Por otro lado, el cliente *asume* que la implementación del TAD es correcta y no se preocupa por los detalles. Cuando estamos usando una de las clases preincorporadas de Java, podemos darnos el lujo de pensar exclusivamente como clientes.

Cuando implementamos un TAD, en cambio, debemos escribir código cliente para probarlo. En este caso, a veces hace falta pensar cuidadosamente qué rol estamos jugando en un instante dado.

En las siguientes secciones vamos a cambiar de rol e investigar una manera de implementar el TAD Pila usando un arreglo. Es hora de empezar a pensar como un proveedor.

15.12 Implementación del TAD Pila usando arreglos

Las variables de instancia para esta implementación son un arreglo de Objects, que contendrá los elementos de la pila, y un índice entero que llevará la cuenta de cuál es el siguiente espacio disponible en el arreglo. Inicialmente, el arreglo está vacío y el índice es 0.

Para agregar un elemento a la pila (push), vamos a copiar una referencia a él en la pila e incrementar el índice. Para quitar un elemento (pop) tenemos que decrementar el índice primero y después copiar el elemento afuera.

Esta es la definición de la clase:

```
public class Stack {
    Object[] arreglo;
    int indice;

    public Stack () {
        this.arreglo = new Object[128];
        this.indice = 0;
    }
}
```

Como de costumbre, una vez que elegimos las variables de instancia, es un proceso mecánico el de escribir un constructor. Por ahora, el tamaño por defecto es de 128 ítems. Después vamos a considerar mejores formas de manejar esto.

Chequear si la pila está vacía es trivial:

```
public boolean isEmpty () {  
    return indice == 0;  
}
```

Es importante recordar, sin embargo, que el número de elementos en la pila no es el mismo que el tamaño del arreglo. Inicialmente el tamaño es 128, pero el número de elementos es 0.

Las implementaciones de push y pop surgen naturalmente de la especificación.

```
public void push (Object elem) {  
    arreglo[indice] = elem;  
    indice++;  
}  
  
public Object pop () {  
    indice--;  
    return arreglo[indice];  
}
```

Para probar estos métodos, podemos sacar ventaja del código cliente que usamos en el ejercicio de la Stack preincorporada. Todo lo que tenemos que hacer es comentar la línea `import java.util.Stack`. Después, en vez de usar la implementación de la pila de `java.util` el programa va a usar la implementación que acabamos de escribir.

Si todo va de acuerdo al plan, el programa debería funcionar sin ningún cambio adicional. Otra vez, una de las fortalezas de usar un TAD es que es posible cambiar las implementaciones sin cambiar el código cliente.

15.13 Redimensionando el arreglo

Una debilidad de esta implementación es que elige un tamaño arbitrario para el arreglo cuando la Stack es creada. Si el usuario apila más de 128 ítems, causará una excepción de tipo `ArrayIndexOutOfBoundsException`.

Una alternativa es dejar que el código cliente especifique el tamaño del arreglo. Esto alivia el problema, pero requiere que el cliente sepa de antemano cuántos ítems va a necesitar, y eso no siempre es posible.

Una solución mejor es verificar si el arreglo está lleno y agrandarlo si es necesario. Ya que no tenemos idea de qué tan grande tiene que ser el arreglo, una estrategia razonable es empezar con un tamaño pequeño y

duplicarlo cada vez que se sobrepasa. Acá tenemos una versión mejorada de push:

```
public void push (Object item) {
    if (lleno ()) redimensionar ();

    // en este punto podemos probar
    // que indice < arreglo.length

    arreglo[indice] = item;
    indice++;
}
```

Antes de poner un nuevo ítem en el arreglo, debemos chequear si el arreglo está lleno. En ese caso, llamamos a redimensionar. Después del if, sabemos que o bien (1) ya había tamaño en el arreglo, o (2) el arreglo fue redimensionado y ahora sí hay espacio. Si `lleno` y `redimensionar` son correctos, entonces podemos probar que `indice < arreglo.length`, y de esta manera, la siguiente sentencia no causará ninguna excepción. Ahora, todo lo que tenemos que hacer es implementar `lleno` y `redimensionar`.

```
private boolean lleno () {
    return indice == arreglo.length;
}

private void redimensionar () {
    Object[] nuevoArreglo = new Object[arreglo.length * 2];

    // asumimos que el arreglo anterior estaba lleno
    for (int i=0; i<arreglo.length; i++) {
        nuevoArreglo[i] = arreglo[i];
    }
    arreglo = nuevoArreglo;
}
```

Ambos métodos son declarados `private`⁴, lo cual significa que no pueden ser llamados desde otra clase, sólo desde esta. Esto es aceptable, ya que no hay razón para el código cliente de usar estas funciones, y a la vez es deseable, ya que refuerza la barrera entre el código proveedor y el cliente.

La implementación de `lleno` es trivial; simplemente chequea que si el índice ha pasado más allá del rango de índices válidos.

4. N.d.T.: Privados.

La implementación de redimensionar es directa, con la salvedad de que asume que el arreglo viejo está lleno. En otras palabras, esta presunción es una precondition de este método. Es simple ver que esta precondition es satisfecha, ya que la única manera de llamar a redimensionar es si `lleno` dio verdadero, lo cual sólo puede suceder si `indice` es igual a `arreglo.length`.

Al final de redimensionar, reemplazamos el arreglo viejo con el nuevo (causando que el viejo sea reclamado por el recolector de basura). La nueva `arreglo.length` es el doble de grande que la anterior, e `indice` no ha cambiado, por lo que ahora debe ser cierto que `index < arreglo.length`. Esta aseveración es una **postcondición** de redimensionar: algo que debe ser cierto cuando este método está completo (siempre y cuando las precondiciones hayan sido satisfechas).

Precondiciones, postcondiciones e invariantes son herramientas útiles para analizar programas y demostrar su corrección. En este ejemplo hemos demostrado un estilo de programación que facilita el análisis de programas y un estilo de documentación que ayuda a demostrar corrección.

15.14 Glosario

tipo de dato abstracto (TAD): Tipo de dato (usualmente una colección de objetos) que está definido por un conjunto de operaciones, pero que puede ser implementado en una variedad de formas.

cliente: Programa que usa un TAD (o persona que escribió el programa).

proveedor: Código que implementa un TAD (o persona que lo escribió).

clase adaptadora: Una de las clases de Java, como `Double` e `Integer` que provee objetos para contener tipos primitivos y métodos que operan en ellos.

private: Palabra clave de Java que indica que un método o variable de instancia no puede ser accedido desde el exterior de la definición de la clase misma.

notación infija: Forma de escribir expresiones matemáticas con los operadores entre los operandos.

notación polaca: Forma de escribir expresiones matemáticas con los operadores después de los operandos.

parser: Leer una cadena de caracteres o tokens y analizar su estructura gramatical.

token: Conjunto de caracteres que son tratados como una unidad a los efectos del parseado, como las palabras en un lenguaje natural.

delimitador: Carácter que es usado para separar tokens, como la puntuación en un lenguaje natural.

predicado: Sentencia matemática que es o bien verdadera o bien falsa.

postcondición: Predicado que debe ser cierto al final de un método (asumiendo que las precondiciones fueran ciertas al comienzo).

15.15 Ejercicios

Ejercicio 15.1

Escribir un método llamado `reverso` que toma un arreglo de enteros y recorre el arreglo apilando cada item en una pila, y después desapila los items de la misma, poniéndolos otra vez en el arreglo en el orden inverso al original.

El objetivo de este ejercicio es practicar los mecanismos para crear adaptadores, apilar y desapilar objetos y castear objetos a otros de un tipo específico.

Ejercicio 15.2

Este ejercicio está basado en la solución del Ejercicio 14.1. Empezar haciendo una copia de la implementación de `ListaInt` llamada `ListaEnlazada`.

- Transformar la implementación de la lista enlazada haciendo el contenido un `Object` en vez de un entero. Modificar el código de testeo de manera acorde y correr el programa.
- Escribir un método llamado `dividir` para la `ListaEnlazada` que toma una `String`, la parte en palabras (usando espacios como delimitadores), y devuelve una lista de `Strings`, con una palabra por nodo. Tu implementación debería ser eficiente, de manera que el tiempo que tome sea proporcional al número de palabras en la cadena.
- Escribir un método `unir` para la `ListaEnlazada` que devuelve una cadena que contiene la representación en tipo `String` de cada uno de los objetos de la lista, en el orden en que aparecen, con espacios entre cada uno.
- Escribir un método `toString` para `ListEnlazada`.

Ejercicio 15.3

Escribir una implementación del TAD Pila usando tu propia implementación de `ListaEnlazada` como estructura de datos subyacente. Hay dos enfoques comunes para esto: la Pila puede contener a la `ListaEnlazada` como variable de instancia, o la clase Pila puede extender la clase `ListaEnlazada`. Elegí la que te parezca

mejor o, si te sentís ambicioso, implementá ambas y comparalas.

Ejercicio 15.4

Escribir un programa llamado `Balance.java` que lee un archivo y verifica que los paréntesis `()` y corchetes `[]` y llaves `{}` están balanceadas y anidadas correctamente. PISTA: Ver la Sección C.3 para saber cómo leer un archivo.

Ejercicio 15.5

Escribir un método llamado `evaluarPostfijo` que toma un `String` conteniendo una expresión postfija y devuelve un `double` que contiene el resultado. Podés usar un `StringTokenizer` para parsear el `String` y una Pila de `Double` para evaluar la expresión.

Ejercicio 15.6

Escribir un programa que pide al usuario una expresión matemática postfija y que evalúa la expresión y luego imprime el resultado. Los próximos pasos son mi sugerencia para el plan de desarrollo del programa.

- a. Escribir un programa que le pide al usuario una entrada e imprime la cadena ingresada, una y otra vez, hasta que el usuario escribe “salir”. Mirar la Sección C.2 para más información sobre cómo obtener entrada del teclado. Podés usar el siguiente código como punto inicial:

```
public static void cicloEntrada() throws IOException {
    BufferedReader stdin =
        new BufferedReader(new InputStreamReader(System.in));

    while (true) {
        System.out.print ("=>");    // imprimir un indicador
        String s = stdin.readLine(); // obtener la entrada
        if (s == null) break;
        // verificar si s es "salir"
        // imprimir s
    }
}
```

- b. Identificar métodos auxiliares que te parezcan que pueden servir, escribirlos y depurarlos por separado. Sugerencias posibles son: `esOperador`, `esOperando`, `parsearExpresion`, `hacerOperacion`.
- c. Sabemos que queremos guardar valores `int` en la pila y luego poder sacarlos, lo cual significa que tendremos que usar una clase wrapper. Asegurate de saber cómo hacerlo, y probá esas operaciones por separado. Quizás, podés agregarles métodos que faciliten la interfaz.
- d. Escribir una versión de `evaluar` que sólo soporta un tipo de operador (como la suma). Probala por separado.

- e. Conectar tu evaluador con tu ciclo de entrada de expresiones.
- f. Agregar las otras operaciones.
- g. Una vez que tengas un código que funciona, podrías querer evaluar el diseño estructural. ¿Cómo deberías dividir el código en clases? ¿Qué variables de instancia debe tener cada una de estas clases? ¿Qué parámetros deberían ser pasados?
- h. Además de hacer elegante el diseño, deberías también hacer que el código sea robusto, es decir, que no deberían provocarse excepciones no manejadas, incluso si el usuario ingresa algo raro.

Capítulo 16

Colas y colas de prioridad

Este capítulo presenta dos TADs: Colas y Colas de Prioridad. En la vida real, una **cola** es una fila de clientes esperando por algún servicio. En la mayoría de los casos, el primer cliente en la fila es el cliente más próximo a ser atendido. Sin embargo, hay algunas excepciones. Por ejemplo, en los aeropuertos, aquellos clientes cuyos vuelos están por partir son a veces atendidos a pesar de encontrarse a la mitad de la fila. También, en los supermercados, un cliente amable puede dejar pasar adelante a otro con sólo unos pocos productos.

La regla que determina quién es atendido a continuación se llama **disciplina de la cola**. La más simple es conocida como **FIFO**, del inglés “first-in-first-out”¹. La disciplina más general, es la que se conoce como **encolado con prioridad**, en la cual a cada cliente se le asigna una prioridad, y el cliente con la prioridad más alta va primero, sin importar el orden de arribo. La razón por la cual digo que esta es la disciplina más general es que la prioridad puede ser basada en cualquier cosa: a qué hora sale el avión, cuántos productos tiene el cliente, o qué tan importante es el cliente. Desde luego, no todas las disciplinas de cola son “justas”, pero la justicia se encuentra en el ojo del espectador.

El TAD Cola y el TAD Cola de Prioridad tienen el mismo conjunto de operaciones y sus interfaces son la misma. La diferencia se encuentra en la semántica de las operaciones: una cola usa una política FIFO, mientras que la Cola de Prioridad (como el nombre sugiere) usa una política de encolado con prioridades.

Como con la mayoría de los TADs, hay muchas formas de implementar colas. Dado que la cola es una colección de elementos, podemos usar cualquiera de los mecanismos básicos para almacenar elementos, incluyendo

1. N.d.T.: “El primero en entrar es el primero en salir”.

arreglos y listas. Nuestra elección de cuál usaremos se basará en la eficiencia —cuánto toma efectuar las operaciones que queremos utilizar— y, en parte, en la facilidad de la implementación.

16.1 El TAD Cola

El TAD cola se define por las siguientes operaciones:

constructor: Crear una nueva cola, vacía.

agregar: Agregar un elemento a la cola.

quitar: Quitar un elemento de la cola y devolverlo. El elemento que se devuelve es el primero en haber sido agregado.

estaVacía: Verifica si la cola está vacía.

Esta es una implementación de una Cola genérica, basada en la clase preincorporada `java.util.LinkedList`:

```
public class Cola {
    private LinkedList lista;

    public Cola() {
        lista = new LinkedList ();
    }

    public boolean estaVacía() {
        return lista.isEmpty();
    }

    public void agregar(Object obj) {
        lista.addLast (obj);
    }

    public Object quitar() {
        return list.removeFirst ();
    }
}
```

Un objeto cola contiene una única variable de instancia, que es la lista que la implementa. Para cada uno de los métodos, todo lo que tenemos que hacer es llamar a un método de la clase `LinkedList`.

16.2 Veneer

Al utilizar una `LinkedList` para implementar una Cola, podemos aprovechar el código ya existente; el código que escribimos sólo traduce métodos de `LinkedList` en métodos de Cola. Una implementación así se llama **veneer**². En la vida real, el enchapado (veneer en inglés) es una fina capa de un madera de buena calidad que se utiliza en la manufactura de muebles para esconder madera de peor calidad por debajo. Los científicos de la computación utilizan esta metáfora para describir una pequeña porción de código que oculta los detalles de implementación y proveen una interfaz más simple o más estándar.

El ejemplo de la Cola demuestra una de las cosas interesantes de un veneer, y es que es fácil de implementar, y uno de los peligros, que es el **riesgo de eficiencia**.

Normalmente, cuando llamamos a un método, no nos preocupan los detalles de su implementación. Pero hay un “detalle” que sí querríamos conocer—la eficiencia del método. ¿Cuánto le toma ejecutarse, en función de la cantidad de elementos de la lista?

Para responder esta pregunta, debemos saber más acerca de la implementación. Si asumimos que `LinkedList` está efectivamente implementada como una lista enlazada, entonces la implementación de `removeFirst`³ probablemente se vea parecido a esto:

```
public Object removeFirst () {
    Object resultado = cabeza;
    cabeza = cabeza.siguiente;
    return resultado.carga;
}
```

Asumimos que `cabeza` referencia al primer nodo de la lista, y que cada nodo contiene una carga y una referencia al siguiente nodo de la lista.

No hay ciclos, ni llamadas a funciones aquí, con lo que el tiempo de ejecución de este método, es más o menos el mismo, cada vez. A un método así se lo conoce como un método de **tiempo constante**.

La eficiencia del método `addLast`⁴ es muy diferente. He aquí una implementación hipotética:

```
public void addLast (Object obj) {
    // caso especial: lista vacía
    if (cabeza == null) {
```

2. N.d.T.: En inglés significa “enchapado”.

3. N.d.T.: En inglés significa quitarPrimero.

4. N.d.T.: En inglés significa agregarAlFinal.


```

        cabeza = new Nodo(obj, null);
        return;
    }
    Nodo ultimo;
    for (ultimo = cabeza; ultimo.siguiiente != null;
        ultimo = ultimo.siguiiente) {
        // atravesar la lista para encontrar el último nodo
    }
    ultimo.siguiiente = new Nodo(obj, null);
}

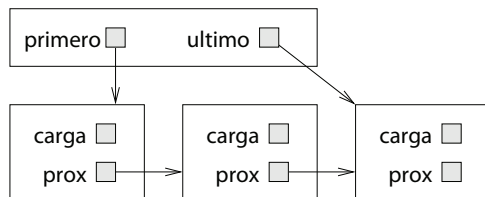
```

El primer condicional controla el caso especial de añadir un nuevo nodo en una lista vacía. En ese caso, nuevamente, el tiempo de ejecución no depende del largo de la lista. Sin embargo, en el caso general, debemos atravesar la lista para encontrar el último elemento de modo que podamos hacer que referencia al nuevo nodo.

Esta iteración toma un tiempo proporcional al largo de la lista. Dado que el tiempo de ejecución es una función lineal del largo de la lista, decimos que este método es de **tiempo lineal**. Comparado con tiempo constante, es muy malo.

16.3 Cola enlazada

Queríamos una implementación del TAD cola que pueda efectuar todas las operaciones en tiempo constante. Una forma de lograr esto es implementando una **cola enlazada**, que es similar a la lista enlazada en el sentido de que se compone de cero o más objetos `Nodo` enlazados. La diferencia es que la cola mantiene una referencia tanto al primero como al último nodo, como se muestra en la figura.



Así es cómo se vería una implementación de una Cola enlazada:

```

public class Cola {
    public Nodo primero, ultimo;

    public Cola () {
        primero = null;
        ultimo = null;
    }
    public boolean estaVacia() {
        return primero == null;
    }
}

```

Hasta acá es directa la implementación. En una cola vacía, tanto primero como ultimo son null. Para verificar si la lista está vacía, sólo tenemos que controlar uno sólo. El método agregar es un poco más complicado, porque tenemos que manejar varios casos especiales.

```

    public void agregar(Object obj) {
        Nodo nodo = new Nodo(obj, null);
        if (ultimo != null) {
            ultimo.siguiente = nodo;
        }
        ultimo = nodo;
        if (primero == null) {
            primero = ultimo;
        }
    }
}

```

La primera condición verifica que ultimo referencie algún nodo; si es así, debemos hacer que ese nodo referencie al nodo nuevo. La segunda condición maneja el caso especial de que la lista haya estado inicialmente vacía. En este caso, tanto primero como ultimo deben referenciar al nuevo nodo. El método quitar también controla varios casos especiales.

```

    public Object quitar() {
        Nodo resultado = primero;
        if (primero != null) {
            primero = primero.siguiente;
        }
        if (primero == null) {
            ultimo = null;
        }
        return resultado;
    }
}

```

La primera condición verifica si había algún nodo en la cola. Si así fuera, debemos hacer que el primero referencie al siguiente. La segunda condición controla el caso de que la lista ahora está vacía, en cuyo caso debemos hacer que ultimo valga también null.

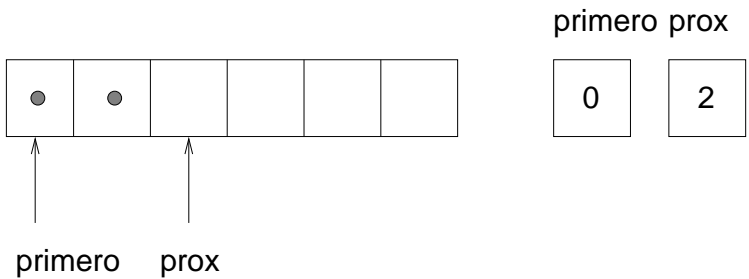
Como ejercicio, dibujá diagramas mostrando estas operaciones tanto en el caso normal, como en los casos especiales, y convencete de que son correctas.

Claramente, esta implementación es más complicada que la veneer, y es más difícil de demostrar que es correcta. La ventaja es que hemos alcanzado el objetivo: agregar y quitar son operaciones de tiempo constante.

16.4 Buffer circular

Otra implementación común de una cola es un **buffer circular**. La palabra “buffer” es un nombre general para un espacio de almacenamiento temporal, a pesar de que muchas veces se refiere a un arreglo, como ocurre en este caso. Qué quiere decir que sea “circular” debería quedar claro en breve.

La implementación de un buffer circular es similar a la implementación de la pila en la Sección 15.12. Los elementos de la cola se almacenan en un arreglo, y usamos índices para mantener un registro de dónde estamos ubicados dentro del arreglo. En la implementación de la pila, había un único índice que apuntaba al siguiente espacio disponible. En la implementación de cola, hay dos índices: primero apunta al espacio del arreglo que contiene el primer cliente en cola y prox apunta al siguiente espacio disponible. La siguiente figura muestra una cola con dos elementos (representados por puntos).



Hay dos maneras de pensar las variables primero y ultimo. Literalmente, son enteros, y sus valores se muestran en las cajas a la derecha. Abstractamente, sin embargo, son índices del arreglo, y por lo tanto, sue-

len dibujarse como flechas apuntando a ubicaciones en el arreglo. La representación de flecha es conveniente, pero deberías recordar que los índices no son referencias; son sólo enteros. Aquí hay una implementación incompleta de una cola sobre arreglo:

```
public class Cola {
    public Object[] arreglo;
    public int primero, prox;

    public Queue () {
        arreglo = new Object[128];
        primero = 0;
        prox = 0;
    }

    public boolean estaVacia () {
        return primero == prox;
    }
}
```

Las variables de instancia y el constructor salen de manera directa, sin embargo, nuevamente tenemos el problema de elegir un tamaño arbitrario para el arreglo. Más tarde resolveremos el problema, como hicimos con la pila, redimensionándolo si el arreglo se llena.

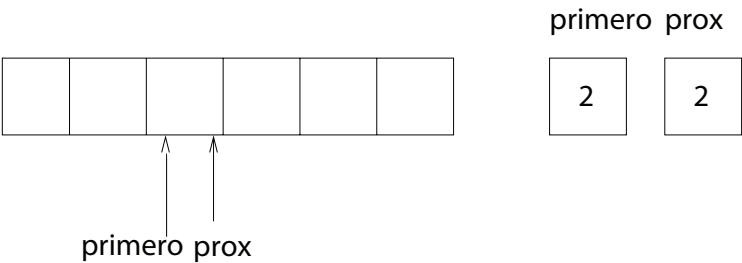
La implementación de `estaVacia` es algo sorprendente. Podrías haber pensado que `primero == 0` hubiera indicado una cola vacía, pero eso ignora el hecho de que la cabeza de la cola no está necesariamente al inicio del arreglo. En cambio, sabemos que la cola está vacía si `primero` es igual a `prox`, en cuyo caso no hay más elementos. Una vez que veamos la implementación de `agregar` y `quitar`, esta condición tendrá más sentido.

```
public void agregar (Object elemento) {
    arreglo[prox] = elemento;
    prox++;
}

public Object quitar () {
    Object resultado = arreglo[primero];
    primero++;
    return resultado;
}
```

El método `agregar` se parece mucho a `apilar` en la Sección 15.12; pone un nuevo elemento en el próximo espacio disponible y luego incrementa el índice. El método `quitar` es similar. Toma el primer elemento de la cola,

y luego incrementa primero de modo que referencie a la nueva cabeza de la cola. La siguiente figura muestra cómo se vería la cola luego de que ambos items se hayan quitado.



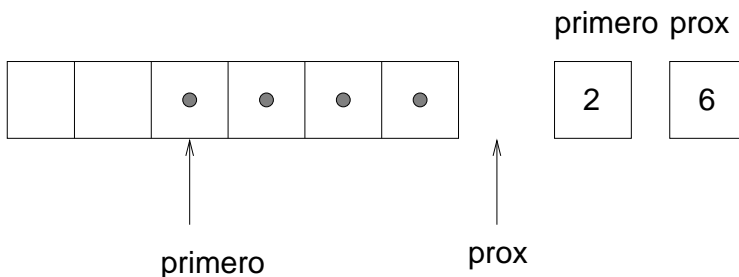
Siempre es cierto que prox apunta a un espacio disponible. Si primero alcanza a prox y apuntan al mismo espacio, entonces primero está apuntando a una posición “vacía”, y la cola está vacía. Pongo “vacía” entre comillas porque es posible que la ubicación a la que apunta primero en realidad contenga un valor (no hacemos nada para asegurarnos de que las posiciones vacías contengan null); por otro lado, dado que sabemos que la cola está vacía, nunca vamos a leer esta posición, de modo que podemos pensarla, abstractamente, como si estuviera vacía.

Ejercicio 16.1

Modificar `quit` para que devuelva `null` si la cola está vacía.

El otro problema de esta implementación es que eventualmente se quedará sin espacio. Cuando agregamos un elemento incrementamos `prox` y cuando quitamos un elemento incrementamos `primero`, pero nunca decrementamos ninguno de los dos. ¿Qué pasa cuando llegamos al final del arreglo?

La siguiente figura muestra la cola después de que agregamos cuatro elementos más:



El arreglo ahora está lleno. No hay “próximo espacio disponible,” con lo que `prox` no tiene a dónde apuntar. Una posibilidad es redimensionar el arreglo, como hicimos con la implementación de la pila. Pero en ese caso el arreglo sólo continuaría creciendo independientemente de cuántos elementos haya efectivamente en la cola. Una mejor solución es dar la vuelta hasta el principio del arreglo y reutilizar los espacios disponibles ahí. Esta “vuelta” es la razón por la cual se conoce a esta implementación como *buffer circular*.

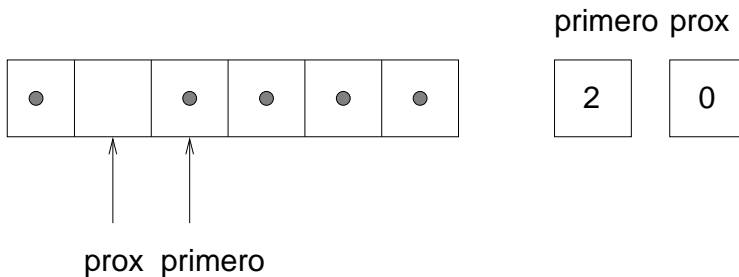
Una forma de reiniciar el índice es agregar un caso especial cuando lo incrementamos:

```
prox++;
if (prox == arreglo.length) prox = 0;
```

Una alternativa más vistosa es utilizar el operador módulo:

```
prox = (prox + 1) % arreglo.length;
```

En cualquier caso, tenemos un último problema por resolver. ¿Cómo sabemos si la cola está *realmente* llena, con la implicación de que no podemos agregar un nuevo elemento? La siguiente figura muestra cómo se vería la cola cuando esté “llena”.



Hay aún un espacio disponible en el arreglo, pero la cola está llena porque si agregamos otro elemento, entonces tendríamos que incrementar `prox` de modo tal que `prox == primero`, en cuyo caso ¡parecería que la cola estuviera vacía!

Para evitar eso, sacrificamos un espacio en el arreglo. Entonces, ¿cómo sabemos si la cola está llena?

```
if ((prox + 1) % arreglo.length == primero)
```

¿Y qué deberíamos hacer si el arreglo está lleno? En ese caso, redimensionar el arreglo es probablemente la única opción.

Ejercicio 16.2

Escribir una implementación de una cola utilizando un buffer circular que se redimensiona cuando sea necesario.

16.5 Cola de prioridad

El TAD Cola de Prioridad tiene las mismas interfaces que el TAD Cola, pero semántica diferente. La interface es:

constructor: Crear una cola nueva vacía.

agregar: Agregar un nuevo elemento a la cola.

quitar: Quitar y devolver un elemento de la cola. El elemento que se devuelve es aquel con la más alta prioridad.

estaVacía: Verificar si la cola está vacía.

La diferencia semántica es que el elemento que se quita de la cola no es necesariamente el primero que se agregó. Más bien, es cualquiera que tenga la máxima prioridad. Cuáles son las prioridades y cómo se comparan entre sí, no lo especifica la implementación de la Cola de Prioridad. Depende de qué elementos están en la cola.

Por ejemplo, si los elementos de la cola tienen nombres, podríamos elegirlos en orden alfabético. Si son puntajes de bowling, podríamos elegirlos de mayor a menor, pero si son puntajes de golf, los elegiríamos de menor a mayor.

Con lo que nos encontramos con un nuevo problema. Querríamos una implementación de la Cola de Prioridad que sea genérica—debería funcionar con cualquier tipo de objeto—pero al mismo tiempo el código de la Cola de Prioridad debe ser capaz de comparar los objetos que contiene.

Hemos visto una forma de implementar estructuras de datos genéricas usando `Object`, pero eso no resuelve este problema, porque no hay forma de comparar dos `Object` a menos que sepamos de qué tipo son. La respuesta yace en una característica de Java que se llama **metaclase**.

16.6 Metaclase

Una metaclase es un conjunto de clases que proveen un conjunto común de métodos. La definición de la metaclase especifica los requerimientos que debe cumplir una clase para ser miembro del conjunto.

A menudo las metaclases tienen nombres que terminan en “able” para indicar la capacidad fundamental que requiere dicha metaclase. Por ejemplo, cualquier clase que provee un método llamado `draw` puede ser miembro de la metaclase llamada `Drawable`⁵. Cualquier clase que contiene un método `start`⁶ puede ser miembro de la metaclase `Runnable`⁷.

Java provee una metaclase preincorporada que podemos usar en una implementación de una Cola de Prioridad. Se llama `Comparable`, y significa precisamente eso. Cualquier clase que pertenezca a la metaclase `Comparable` debe proveer un método llamado `compareTo` que compara dos objetos y devuelve un valor indicando si uno es más grande o más chico que el otro, o si son iguales.

Muchas de las clases preincorporadas son miembros de la metaclase `Comparable`, incluyendo las clases que encapsulan números como `Integer` o `Double`.

En la próxima sección te mostraré cómo escribir un TAD que manipula una metaclase. Luego, veremos cómo escribir una nueva clase que pertenezca a una metaclase preexistente. En el próximo capítulo veremos cómo definir una nueva metaclase.

16.7 Implementación de Cola de Prioridad sobre Arreglo

En la implementación de la Cola de Prioridad, toda vez que especifiquemos el tipo de los elementos de la cola, debemos escribir la metaclase `Comparable`. Por ejemplo, las variables de instancia son un arreglo de `Comparables` y un entero:

```
public class ColaDePrioridad {  
    private Comparable[] arreglo;  
    private int indice;  
}
```

Como es usual, `indice` es el índice de la próxima ubicación disponible en el arreglo. Las variables de instancia se declaran `private` de modo que otras clases no puedan tener acceso directo a ellas. El constructor y `estaVacia` son similares a lo que ya hemos visto antes. El tamaño inicial del arreglo es arbitrario.

5. N.d.T.: Dibujable.

6. N.d.T.: iniciar.

7. N.d.T.: Ejecutable


```

public ColaDePrioridad() {
    arreglo = new Comparable [16];
    indice = 0;
}

public boolean estaVacia() {
    return indice == 0;
}

```

El método agregar es similar a apilar:

```

public void agregar(Comparable elemento) {
    if (indice == arreglo.length) {
        redimensionar();
    }
    arreglo[indice] = elemento;
    indice++;
}

```

El único método substancial en la clase es quitar, que tiene que recorrer el arreglo para encontrar y quitar el elemento más grande:

```

public Comparable quitar () {
    if (indice == 0) return null;

    int maxIndice = 0;

    // buscar el índice del elemento con mayor prioridad
    for (int i=1; i<indice; i++) {
        if (arreglo[i].compareTo (arreglo[maxIndice]) > 0) {
            maxIndice = i;
        }
    }
    Comparable resultado = arreglo[maxIndice];

    // mover el último elemento en el espacio que queda vacío
    indice--;
    arreglo[maxIndice] = arreglo[indice];
    return resultado;
}

```

A medida que recorremos el arreglo, maxIndice guarda el índice del mayor elemento que hayamos visto hasta el momento. Qué significa que sea “más grande” lo determina compareTo. En este caso el método compareTo

lo provee la clase `Integer`, y hace lo que se espera—números más grandes (más positivos) ganan.

16.8 Un cliente de la Cola de Prioridad

La implementación de la Cola de Prioridad está escrita enteramente en términos de objetos `Comparable` pero ¡no existen los objetos de tipo `Comparable`! Adelante, probá crear uno:

```
Comparable comp = new Comparable ();           // ERROR
```

Vas a obtener un mensaje de error en tiempo de compilación que dice algo como: “`java.lang.Comparable` is an interface. It can’t be instantiated”⁸. En Java, las metaclasses se llaman **interfaces**. He evitado usar la palabra hasta ahora porque también significa muchas otras cosas, pero ya es hora de que lo sepas.

¿Por qué no se puede instanciar una metaclass? Porque las metaclasses sólo especifican requerimientos (tenés que tener un método `compareTo`); pero no proveen la implementación.

Para crear un objeto `Comparable`, tenes que crear un objeto que pertenezca al conjunto de `Comparables`, como `Integer`. Entonces podés usar ese objeto en cualquier lugar donde se pida un `Comparable`.

```
ColaDePrioridad cp = new ColaDePrioridad();  
Integer elem = new Integer (17);  
cp.agregar(elem);
```

Este código crea una nueva `ColaDePrioridad` y un nuevo objeto `Integer`. Luego agrega el `Integer` a la cola. El método `agregar` espera un objeto `Comparable` como parámetro, de modo que está perfectamente satisfecho de tomar un `Integer`. Si intentamos pasar un `Rectangle`, el cual no pertenece a `Comparable`, obtenemos un error en tiempo de compilación como “`Incompatible type for method. Explicit cast needed to convert java.awt.Rectangle to java.lang.Comparable`”⁹. Ese es el compilador diciéndonos que si queremos hacer esa conversión debemos hacerla explícita. Podríamos probar hacer lo que dice:

```
Rectangle rect = new Rectangle ();  
cp.agregar((Comparable) rect);
```

8. N.d.T.: `java.lang.Comparable` es una interfaz. No puede ser instanciada.

9. N.d.T.: Tipo incompatible para el método. Se requiere un cast explícito para convertir un `java.awt.Rectangle` en `java.lang.Comparable`.

Pero en ese caso obtenemos un error en tiempo de ejecución, una excepción de tipo `ClassCastException`. Cuando el `Rectangle` trata de pasar como un `Comparable`, el sistema de tiempo de ejecución verifica si satisface los requerimientos, y la rechaza. Con lo que eso es lo que obtenemos por seguir el consejo del compilador.

Para sacar los elementos de la cola, debemos invertir el proceso:

```
while (!cp.estaVacia()) {
    elem = (Integer) cp.quitar();
    System.out.println (elem);
}
```

Este ciclo elimina todos los elementos de la cola y los imprime. Asume que los elementos en la cola son todos `Integer`. Si no, obtendremos una `ClassCastException`.

16.9 La clase Golfista

Por último, veamos cómo podemos hacer que una nueva clase pertenezca a `Comparable`. Como ejemplo de algo con una definición inusual de prioridad más alta, usaremos golfistas:

```
public class Golfista implements Comparable {
    String nombre;
    int puntaje;

    public Golfer (String nombre, int puntaje) {
        this.nombre = nombre;
        this.puntaje = puntaje;
    }
}
```

La definición de clase y el constructor son bastante iguales a los que ya vimos; la diferencia es que tenemos que declarar que `Golfista` implements `Comparable`. En este caso la palabra reservada `implments` significa que `Golfista` implementa la interface especificada por `Comparable`.

Si tratamos de compilar `Golfista.java` en este punto, obtendríamos algo como “class `Golfista` must be declared abstract. It does not define int `compareTo(java.lang.Object)` from interface `java.lang.Comparable`”¹⁰. En otras palabra, para ser un `Comparable`, `Golfista` tiene que proveer el método llamado `compareTo`. De modo que escribamos uno:

10. N.d.T.: “La clase `Golfista` debe ser declarada como abstracta. No define int `compareTo(java.lang.Object)` de la interfaz `java.lang.Comparable`”.

```

public int compareTo (Object obj) {
    Golfista otro = (Golfer) obj;

    int a = this.puntaje;
    int b = otro.puntaje;

    // para los golfistas, más bajo es mejor!
    if (a<b) return 1;
    if (a>b) return -1;
    return 0;
}

```

Dos cosas aquí son un poco sorprendentes. Primero, el parámetro es un `Object`. Eso es porque en general el que llama al método no sabe qué tipo de objetos son los que están siendo comparados. Por ejemplo, en `ColaDePrioridad.java` cuando llamamos al método `compareTo`, pasamos un `Comparable` como parámetro. No necesitamos saber si es un `Integer` o un `Golfista` o lo que fuera.

Dentro de `compareTo` tenemos que convertir el parámetro de un `Object` a un `Golfista`. Como es usual, hay un riesgo cuando hacemos este tipo de casteo: si convertimos al tipo equivocado obtenemos una excepción.

Finalmente, podemos crear algunos golfistas:

```

Golfer tiger = new Golfer ("Tiger Woods", 61);
Golfer phil = new Golfer ("Phil Mickelson", 72);
Golfer hal = new Golfer ("Hal Sutton", 69);

```

Y ponerlos en la cola:

```

cp.agregar(tiger);
cp.agregar(phil);
cp.agregar(hal);

```

Cuando los sacamos:

```

while (!cp.estaVacia()) {
    golfista = (Golfista) cp.quitar();
    System.out.println (golfista);
}

```

Aparecen en orden descendiente (para golfistas):

```

Tiger Woods      61
Hal Sutton       69
Phil Mickelson   72

```

Cuando cambiamos de `Integers` a `Golfistas`, no hemos hecho ningún tipo de cambios en `ColaDePrioridad.java`. Con lo cual, hemos tenido éxito en mantener una barrera entre `ColaDePrioridad` y las clases que la utilizan, permitiéndonos reutilizar el código sin modificación. Más aún, hemos sido capaces de darle al código cliente control sobre la definición de `compareTo`, haciendo esta implementación de `ColaDePrioridad` más versátil.

16.10 Glosario

cola: Conjunto ordenado de objetos esperando por algún servicio.

disciplina de la cola: Las reglas que determinan qué miembro de la cola debe ser quitado a continuación.

FIFO: del inglés “first in, first out,” (“el primero en entrar, es el primero en salir”). Una disciplina de cola en la que el primer miembro en llegar es el primero en ser quitado.

cola de prioridad: Disciplina de cola en la cual cada miembro tiene una prioridad determinada por factores externos. El miembro con mayor prioridad es el primero en ser quitado.

Cola de Prioridad: TAD que define las operaciones que se pueden efectuar en una cola de prioridad.

veneer: Definición de clase que implementa un TAD con definiciones de métodos que son llamadas a otros métodos, a veces, con transformaciones simples. El veneer no hace ningún trabajo significativo, pero mejora o estandariza la interfaz vista por el cliente.

riesgo de performance: Riesgo asociado a un veneer en el cual algunos métodos pueden estar implementados de una forma ineficiente que puede no ser evidente para el cliente.

tiempo constante: Operación cuyo tiempo de ejecución no depende del tamaño de la estructura de datos.

tiempo lineal: Operación cuyo tiempo de ejecución es una función lineal del tamaño de la estructura de datos.

cola enlazada: Implementación de una cola utilizando una lista enlazada que referencia los nodos primero y último.

buffer circular: Implementación de una cola usando un arreglo y los índices del primer elemento y del siguiente espacio disponible.

metaclass: Conjunto de clases. La especificación de la metaclass lista los requerimientos que una clase debe satisfacer para ser incluida en el conjunto.

interface: Nombre que le da Java a la metaclass. No confundir con el significado amplio de la palabra.

16.11 Ejercicios

Ejercicio 16.3

Este ejercicio está basada en el Ejercicio 9.3. Escribir un método `compareTo` para la clase `Racional` que permita a la clase `Racional` implementar la interface `Comparable`. Pista: no olvidar que el parámetro es un `Object`.

Ejercicio 16.4

Escribir una definición de clase para `ListaOrdenada`, que extienda la clase `ListaEnlazada`. Una `ListaOrdenada` es similar a una `ListaEnlazada`; la diferencia es que los elementos deben ser `Comparables` y la lista está ordenada en orden decreciente.

Escribir un método de objeto para `ListaOrdenada` llamado `agregar` que toma un `Comparable` como parámetro y que agrega el nuevo objeto a la lista, en la posición apropiada de modo que la lista permanezca ordenada.

Si querés, podés escribir un método auxiliar en la clase `Nodo`.

Ejercicio 16.5

Escribir un método de objeto para la clase `ListaOrdenada` llamado `maximo` que puede ser llamado en un objeto `ListaOrdenada`, y que devuelva la carga más grande en la lista, o `null` si la lista está vacía.

Podés asumir que todos los elementos en la carga de los nodos pertenecen a una clase que pertenece a la metaclass `Comparable`, y que todo par de elementos puede ser comparado.

Ejercicio 16.6

Escribir una implementación de una Cola de Prioridad usando una lista enlazada. Hay dos maneras en que podrías proceder:

- Una Cola de Prioridad podría contener un objeto `ListaEnlazada` como variable de instancia.
- Una Cola de Prioridad podría contener una referencia al primer objeto `Nodo` en la lista enlazada.

Pensá acerca de los pros y las contras de cada uno y elegí uno. Además podés elegir si mantener la lista ordenada (agregado lento, quitado rápido) o desordenada (quitado lento, agregado rápido).

Ejercicio 16.7

Una cola de eventos es una estructura de datos que guarda un conjunto de eventos, donde cada evento tiene una hora asociada a él. El TAD es:

constructor: crear una nueva cola de eventos vacía.

agregar: insertar un nuevo evento en la cola. Los parámetros son el evento, que es un `Object`, y la hora a la que el evento ocurre, que es un objeto `Date`¹¹. El evento no debe ser `null`.

proximaHora: devuelve la fecha (el objeto `Date`) en la cual ocurre el próximo evento, donde el “próximo” evento es aquel en la cola con una hora más próxima. No quitar el evento de la cola. Devolver `null` si la cola está vacía.

proximoEvento: devuelve el próximo evento (un `Object`) de la cola y lo quita. Devuelve `null` si la cola está vacía.

La clase `Date` está definida en `java.util` e implementa `Comparable`. De acuerdo a la documentación su `compareTo` devuelve “el valor 0 si el `Date` pasado como parámetro es igual a este `Date`; un valor menor a 0 si este `Date` es anterior al `Date` parámetro; y un valor mayor a 0 si este `Date` es posterior al `Date` parámetro.”

Escribir una implementación de una cola de eventos utilizando el TAD `ColaDePrioridad`. No deberías hacer ningún tipo de suposición sobre cómo se implementa la `ColaDePrioridad`.

PISTA: creá una clase llamada `Evento` que contenga una `Date` y un `Object` evento, y que implemente `Comparable` apropiadamente.

11. N.d.T.: Fecha.

Capítulo 17

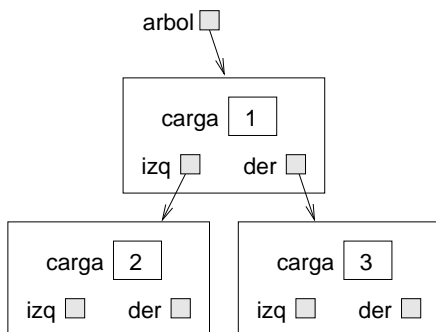
Árboles

17.1 Un nodo del árbol

Al igual que las listas, los árboles se componen de nodos. Un tipo común de árbol es un **árbol binario**, en el que cada nodo contiene una referencia a otros dos nodos (que pueden ser nulos). La definición de la clase tiene el siguiente aspecto:

```
public class Arbol {  
    Object carga;  
    Arbol izq, der;  
}
```

Al igual que los nodos de la lista, los nodos del árbol contienen carga: en este caso un `Object` genérico. Las otras variables de instancia se llaman `izq` y `der`, de acuerdo a una forma estándar para representar árboles gráficamente:



La parte superior del árbol (el nodo referenciado por `arbol`) es llamado **raíz**. De acuerdo con la metáfora del árbol, los otros nodos se llaman ramas y los nodos en los extremos con referencias nulas se llaman **hojas**. Puede parecer extraño que lo dibujemos con la raíz en la parte superior y las hojas en la parte inferior, pero eso no es lo más extraño.

Para empeorar las cosas, científicos de la computación mezclaron otra metáfora: el árbol genealógico. El nodo superior se llama a veces nodo **padre** y los nodos a los que referencia son sus nodos **hijos**. Los nodos con el mismo padre se llaman **hermanos**, etcétera.

Por último, también hay un vocabulario geométrico para referirse a los árboles. Ya he mencionado izquierda y derecha, pero también está “arriba” (hacia el padre/raíz) y “abajo” (hacia el hijo/hoja). Además, todos los nodos que están a la misma distancia desde la raíz comprenden un mismo **nivel** del árbol.

No sé por qué necesitamos tres metáforas para hablar de los árboles, pero están.

17.2 Construcción de árboles

El proceso de montaje de los nodos del árbol es similar al proceso de montaje de las listas. Tenemos un constructor de los nodos del árbol que inicializa las variables de instancia.

```
public Arbol (Object carga, Arbol izq, Arbol der) {
    this.carga = carga;
    this.izq = izq;
    this.der = der;
}
```

Asignamos a los nodos hijos en primer lugar:

```
Arbol izq = new Arbol (new Integer(2), null, null);
Arbol der = new Arbol(new Integer(3), null, null);
```

Podemos crear el nodo padre y enlazarlo a los hijos al mismo tiempo:

```
Arbol arbol = new Arbol (new Integer(1), izq, der);
```

Este código produce el estado que se muestra en la figura anterior.

17.3 Recorrido de árboles

La forma más natural de recorrer un árbol es la recursiva. Por ejemplo, para sumar todos los números enteros de un árbol, podemos escribir el siguiente método de clase:

```

public static int total (Arbol arbol) {
    if (arbol == null) return 0;
    Integer carga = (Integer) arbol.carga;
    return carga.intValue() + total (arbol.izq) +
                               total (arbol.derecho);
}

```

Este es un método de clase porque nos gustaría usar null para representar un árbol vacío, y hacer de este el caso base para la recursividad. Si el árbol está vacío, el método devuelve 0. De lo contrario hace dos llamadas recursivas para hallar el valor total de sus dos hijos. Por último, suma el valor de su propia carga y devuelve el total.

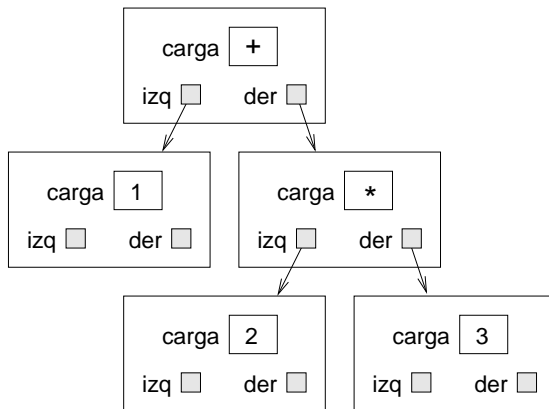
Aunque este método funciona, hay una cierta dificultad para ajustarlo a un diseño orientado a objetos. No debe aparecer en la clase Arbol porque requiere que carga sea un objeto Integer. Si hacemos esa suposición en Tree.java entonces perdemos la ventaja de una estructura de datos genérica.

Por otra parte, este código tiene acceso a las variables de instancia de los nodos Arbol, por lo que “sabe” más de lo que debe sobre la implementación del árbol. Si después modificamos la implementación este código se romperá.

Más adelante en este capítulo vamos a desarrollar maneras de resolver este problema, permitiendo que el código cliente recorra los árboles que contengan cualquier tipo de objetos sin romper la barrera de abstracción entre el código cliente y la implementación. Antes de llegar, vamos a ver una aplicación de los árboles.

17.4 Árboles de expresiones

Un árbol es una forma natural para representar la estructura de una expresión matemática. A diferencia de otras notaciones, puede representar el cálculo de forma inequívoca. Por ejemplo, la expresión infija $1 + 2 * 3$ es ambigua a menos que sepamos que la multiplicación se hace antes de la adición. 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 estos operadores son **binarios**, lo que significa que tienen exactamente dos operandos).

Mirando esta figura, no hay duda de cuál es el orden de las operaciones: la multiplicación sucede en primer lugar con el fin de calcular el primer operando de la suma.

Los árboles de expresiones como este tienen muchos usos. El ejemplo que vamos a ver es la traducción de un formato (postfijo) a otro (infijo). Árboles similares son usados por los compiladores para analizar, optimizar y traducir programas.

17.5 Recorrido

Ya he señalado que la recursividad proporciona una forma natural de recorrer un árbol. Podemos imprimir el contenido de un árbol de esta manera:

```

public static void imprimir (Arbol arbol) {
    if (arbol == null) return;
    System.out.print (arbol.carga + " ");
    imprimir (arbol.izq);
    imprimir (arbol.der);
}

```

En otras palabras, para imprimir un árbol, primero imprime el contenido de la raíz, luego imprime el subárbol izquierdo entero, y luego imprime el subárbol derecho entero. Esta forma de recorrer el árbol se llama **preor-**

der, porque el contenido de la raíz aparece antes que el contenido de sus hijos.

Por ejemplo la expresión de la salida es $1 + 2 * 3$. Esto es diferente de las notaciones postfijo e infijo; y conforma una nueva notación llamada **prefijo**, en donde los operadores aparecen antes que los operandos.

Se puede sospechar que si recorremos el árbol en un orden diferente obtenemos la expresión en diferentes notaciones. Por ejemplo, si imprimimos los subárboles primero, y luego el nodo raíz:

```
public static void imprimirPostorder (Arbol arbol) {
    if (arbol == null) return;
    imprimirPostorder (arbol.izq);
    imprimirPostorder (arbol.der);
    System.out.print (arbol.carga + " ");
}
```

¡Obtenemos la expresión en postfijo $(1 \ 2 \ 3 \ * \ +)$! Como el nombre del método indica, este orden de recorrer es llamado **postorder**. Finalmente, para recorrer un árbol **inorder** imprimimos el subárbol izquierdo, luego la raíz, y después el subárbol derecho:

```
public static void imprimirInorder (Arbol arbol) {
    if (arbol == null) return;
    imprimirInorder (arbol.izq);
    System.out.print (arbol.carga + " ");
    imprimirInorder (arbol.der);
}
```

El resultado es $1 + 2 * 3$, que es la expresión infijo.

Para ser justos, debo señalar que he omitido una complicación importante. A veces cuando escribimos una expresión en infijo debemos usar paréntesis para preservar el orden de las operaciones. Así que un recorrido inorden no es suficiente para generar una expresión infija.

Sin embargo, con algunas mejoras, la expresión de un árbol y los tres recorridos recursivos proporcionan una manera general para traducir de un formato a otro.

17.6 Encapsulamiento

Como mencioné antes, hay un problema en la manera en que se han recorrido los árboles: se rompe la barrera entre el código del cliente (la aplicación que usa el árbol) y el código del proveedor (la implementación de árbol). Idealmente, el código de un árbol debe ser general; no tiene que

saber nada de árboles de expresiones. Y el código que genera y recorre los árboles de expresiones no debe saber sobre la implementación de los árboles. Este criterio de diseño se llama **encapsulamiento de objetos** para distinguirla del encapsulamiento que vimos en la Sección 6.6, que podríamos llamar **encapsulamiento de métodos**.

En la versión actual, el código del `Árbol` sabe demasiado sobre el cliente. En cambio, la clase `Árbol` debe proporcionar la capacidad general de recorrer un árbol de varias maneras. Como recorre, debe realizar las operaciones en cada nodo como fue especificado por el cliente.

Para facilitar esta separación de intereses, crearemos una nueva metaclass, llamada `Visitable`. Los elementos almacenados en un árbol deberán ser visitables, lo que significa que hay que definir un método llamado `visitar` que hace lo que el cliente quiere que se haga en cada nodo. De esta manera el `Árbol` puede realizar el recorrido y el cliente puede realizar las operaciones de los nodos.

Estos son los pasos que tenemos que realizar para mantener una metaclass entre un cliente y un proveedor:

1. Definir una metaclass que especifica los métodos que el código proveedor necesitará llamar en sus componentes.
2. Escribir el código proveedor en términos de la nueva metaclass, en lugar de `Object` genéricos.
3. Definir una clase que pertenezca a la metaclass y que implemente los métodos que son necesarios por el cliente.
4. Escribir el código cliente para utilizar la nueva clase.

Las siguientes secciones muestran estos pasos.

17.7 Definición de una metaclass

Actualmente hay dos formas de implementar una metaclass en Java, como una **interface** o como una **clase abstracta**. La diferencia entre estas no es importante por el momento, por lo que empezaré por definir una interface.

Una definición de la interface se parece mucho a una definición de clase, con dos diferencias:

- La palabra clave `class` es reemplazada por `interface`, y
- La definiciones de los métodos no tienen cuerpo.

Una definición de interface especifica los métodos que la clase tiene que implementar para pertenecer a una metaclass. Las especificaciones incluyen el nombre, el tipo de parámetro, y el tipo de retorno de cada método. La definición de `Visitable` es

```
public interface Visitable {  
    public void visitar ();  
}
```

¡Eso es todo! La definición de `visitar` se parece a cualquier otra definición de método, excepto que no tiene cuerpo. Esta definición especifica que cualquier clase que implemente `Visitable` tiene que tener un método llamado `visitar` que no toma ningún parámetro y devuelve `void`. Como otras definiciones de clase, las definiciones de interface se guardan en un archivo con el mismo nombre de la clase (en este caso `Visitable.java`).

17.8 Implementando una metaclass

Si estamos usando un árbol de expresión para generar una expresión infijo, entonces “visitar” un nodo quiere decir imprimir su contenido. Dado que el contenido de un árbol de expresión son símbolos, vamos a crear una nueva clase llamada `ComponenteLexico`¹ que implementa `Visitable`

```
public class ComponenteLexico implements Visitable {  
    String str;  
  
    public ComponenteLexico (String str) {  
        this.str = str;  
    }  
  
    public void visitar () {  
        System.out.print (str + " ");  
    }  
}
```

Cuando compilemos esta definición de clase (que está en un archivo llamado `ComponenteLexico.java`), el compilador comprobará si los métodos provistos satisfacen los requerimientos especificados por la metaclass. Si no, se producirá un mensaje de error. Por ejemplo, si escribimos mal el nombre de un método que se supone se llame `visitar`, podríamos obtener algo así “class `ComponenteLexico` must be declared abstract. It does not define void `visitar()` from interface `Visitable`”.² Este es uno de los mu-

1. N.d.T.: Token en inglés.

2. N.d.T.: “La clase `ComponenteLexico` debe ser declarada como abstracta. No está definida void `visitar()` en la interfaz `Visitable`”.

chos mensajes de error en el que la solución que propone el compilador está mal. Cuando dice que la clase “debe ser declarada abstracta,” lo que quiere decir es que tenés que arreglar la clase para que implemente la interface correctamente. A veces pienso que habría que golpear a la gente que escribe estos mensajes.

El siguiente paso es modificar el análisis para incorporar objetos de tipo `ComponenteLexico` dentro de los árboles en lugar de Cadenas. Aquí un pequeño ejemplo:

```
String expr = "1 2 3 * +";
StringTokenizer st = new StringTokenizer(expr, " +*/", true);
String token = st.nextToken();
Arbol arbol = new Arbol(new ComponenteLexico(token), null, null);
```

Este código toma el primer componente léxico en la cadena y lo envuelve en un objeto `ComponenteLexico`, luego inserta el `ComponenteLexico` dentro de un nodo del árbol. Si el árbol requiere que la carga sea `Visitable`, se convertirá el `ComponenteLexico` para que sea un objeto `Visitable`. Cuando quitamos el `Visitable` de un árbol, tendremos que convertirlo nuevamente en un `ComponenteLexico`.

Ejercicio 17.1

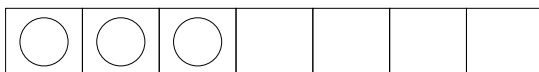
Escribir una versión de `imprimirPreorder` llamada `visitarPreorder` que recorra el árbol e invoque `visitar` en cada nodo en `preorder`.

El flujo de ejecución de métodos como `visitarPreorder` es inusual. El cliente llama a un método proporcionado por la implementación del árbol, y después la implementación del árbol llama a un método proporcionado por el cliente. Este patrón se llama **callback**; es una buena forma de hacer más general el código provisto, sin romper la barrera de abstracción.

17.9 La clase `Vector`

El `Vector` es una clase de Java que ya viene incluida en el paquete `java.util`. Es una implementación de un arreglo de `Object`, con la característica añadida de que puede cambiar automáticamente el tamaño, para que no tengamos que hacerlo nosotros.

Antes de usar la clase `Vector`, tendrás que entender algunos conceptos. Cada `Vector` tiene una capacidad, que es la cantidad de espacio que se ha asignado para almacenar los valores, y un tamaño, que es el número de valores que hay actualmente en el vector. La siguiente figura es un diagrama simple de un `Vector` que contiene tres elementos, pero tiene una capacidad de siete.



Hay dos tipos de métodos para acceder a los elementos de un vector. Proporcionan una semántica distinta y diferentes aptitudes de comprobación de errores, y son fáciles de confundir.

Los métodos más simples de acceso son `get` y `set`, que proporcionan una semántica similar a la del operador índice del array `[]`. `get` toma un índice entero y devuelve el elemento en la posición indicada. `set` toma un índice y un elemento, y almacena el nuevo elemento en la posición indicada, sustituyendo el elemento existente.

Los métodos `get` y `set` no modifican el tamaño del vector (número de elementos). Es responsabilidad del código cliente asegurarse que el vector tiene un tamaño suficiente antes de ser llamado `set` or `get`. El método `size` devuelve el número de elementos que hay en el vector. Si intentas acceder a un elemento que no existe (en este caso los elementos con índices de 3 a 6), obtendrá una excepción `ArrayIndexOutOfBoundsException`.

El otro conjunto de métodos incluye varias versiones de `add` y `remove`. Estos métodos cambian el tamaño del vector y, si es necesario, la capacidad. Una versión de `add` tiene un elemento como parámetro y lo agrega al final del vector. Este método es seguro en el sentido que no causará una excepción.

Otra versión de `add` tiene un índice y un elemento y, como `set`, almacena el nuevo elemento en la posición dada. La diferencia es que `add` no reemplaza un elemento existente; sino que aumenta el tamaño del vector y los elementos se desplazan a la derecha para dejar espacio para uno nuevo. Por lo tanto la llamada `v.add(0, elt)` agrega un nuevo elemento al principio del vector. Por desgracia, este método no es ni seguro ni eficiente; ya que puede causar una excepción `ArrayIndexOutOfBoundsException` y, en la mayoría de las implementaciones, es en tiempo lineal (proporcional al tamaño del vector).

La mayoría de las veces el cliente no tiene que preocuparse por la capacidad. Siempre que el tamaño del Vector cambia, la capacidad se actualiza automáticamente. Por razones de performance, algunas aplicaciones toman el control de esta función, por lo que existen métodos adicionales para aumentar y disminuir la capacidad.

Debido a que el código de cliente no tiene acceso a la implementación de un vector, no está claro cómo se debe recorrer. Por supuesto, una posibilidad es utilizar una variable dentro de un ciclo como índice del vector:


```

for (int i=0; i<v.size(); i++) {
    System.out.println (v.get(i));
}

```

No hay nada malo en ello, pero hay otra forma que sirve para demostrar la clase `Iterator`. Los vectores proporcionan un método llamado `iterator` que devuelve un objeto `Iterator` que permite recorrer el vector.

17.10 La clase `Iterator`

`Iterator` es una interface dentro del paquete `java.util`. Se especifican tres métodos:

`hasNext`: ¿Esta iteración tiene más elementos?

`next`: Devuelve el siguiente elemento, o lanza una excepción si no hay ninguno.

`remove`: Remueve el elemento más reciente de la estructura de datos que estamos recorriendo.

En el ejemplo siguiente se utiliza un iterador para recorrer e imprimir los elementos de un vector.

```

Iterator it = vector.it ();
while (it.hasNext ()) {
    System.out.println (it.next ());
}

```

Una vez que el objeto `Iterator` se crea, es un objeto independiente del `Vector` original. Los cambios posteriores en el `Vector` no se reflejan en el `Iterator`. De hecho, si modifica el `Vector` después de crear un `Iterator` el `Iterator` deja de ser válido. Si accede al `Iterator` de nuevo, causará una excepción `ConcurrentModification`.

En un apartado anterior hemos utilizado la metaclass `Visitable` para permitir al cliente recorrer una estructura de datos sin conocer los detalles de su implementación. Los iteradores proporcionan otra forma de hacerlo 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 puede utilizar y seleccionar elementos uno a la vez (aunque en un orden controlado por el proveedor).

Ejercicio 17.2

Escribir una clase llamada `PreIterator` que implemente la interface `Iterator`, y escriba un método llamado `preorderIterator` para la clase `Arbol` que devuelva

un `PreIterator` que seleccione los elementos de un `Arbol` en preorder. SUGERENCIA: La forma más fácil de crear un iterador es poner elementos en un vector en el orden que desee y luego llamar a `iterator` en el vector.

17.11 Glosario

árbol binario: Árbol en el que cada nodo hace referencia a 0, 1, o 2 nodos dependientes.

raíz: Nodo de nivel superior en un árbol, que no es referenciado por ningún otro nodo.

hoja: Nodo más abajo en un árbol, que no referencia a ningún otro nodo.

padre: Nodo que referencia a un nodo dado.

hijo: Uno de los nodos referido por otro nodo.

nivel: Conjunto de nodos equidistantes de la raíz.

notación prefijo: Forma de escribir una expresión matemática donde cada operador aparece antes de sus operandos.

preorder: Forma de recorrer un árbol, visitando cada nodo antes de sus hijos.

postorder: Forma de recorrer un árbol, visitando a los hijos de cada nodo antes que el nodo mismo.

inorder: Forma de recorrer un árbol, visitando el subárbol izquierdo, a continuación la raíz, luego el subárbol derecho.

variable de clase: Variable `static` declarada fuera de cualquier método. Es accesible desde cualquier método.

operador binario: Operador que toma dos operandos.

encapsulamiento de objetos: Objetivo del diseño de mantener las implementaciones de dos objetos lo más separados posible. Ninguna clase debe conocer los detalles de la implementación de la otra.

encapsulamiento de métodos: Objetivo del diseño de mantener la interfaz de un método independiente de los detalles de su implementación.

callback: Técnica de programación por la cual es posible generalizar una parte de un código abstrayendo un método que se llama en algún punto.

17.12 Ejercicios

Ejercicio 17.3

- ¿Cuál es el valor de la expresión postfijo $1\ 2\ +\ 3\ *$?
- ¿Cuál es la expresión postfijo que es equivalente a la expresión infijo $1\ +\ 2\ *\ 3$?
- ¿Cuál es el valor de la expresión postfijo $17\ 1\ -\ 5\ /\$, asumiendo que $/$ realiza la división entera?

Ejercicio 17.4

La altura de un árbol es el camino más largo desde la raíz hasta cualquier hoja. La altura puede ser definida recursivamente de la siguiente manera:

- La altura de un árbol nulo es 0.
- La altura de un árbol no nulo es $1 + \max(\text{alturaIzq}, \text{alturaDer})$, donde alturaIzq es la altura del hijo de la izquierda y alturaDer es la altura del hijo de la derecha.

Escriba un método llamado `altura` que calcule la altura de un `Arbol` proporcionado como parámetro.

Ejercicio 17.5

Imaginemos que definimos un árbol que contiene objetos comparables como la carga:

```
public class ArbolDeComparable {
    Comparable carga;
    Arbol izq, der;
}
```

Escriba un método de la clase `Arbol` llamada `encontrarMax` que devuelve la mayor carga en el árbol, donde “mayor” se define por `compareTo`.

Ejercicio 17.6

Un árbol de búsqueda binaria es un tipo especial de árbol donde, para cada nodo N :

toda carga en el subárbol izquierdo de $N <$ la carga en el nodo N
y
la carga en el nodo $N <$ toda carga en el subárbol derecho de N

Usando la siguiente definición de clase³, escribir un método de objeto llamado `contains` que toma un `Object` como parámetro y que devuelve `true` si el objeto

3. N.d.T.: `SearchTree` quiere decir árbol de búsqueda.

aparece en el árbol o false en caso contrario. Se puede asumir que el objeto solicitado y todos los objetos en el árbol son de un tipo Comparable.

```
public class SearchTree{
    Comparable carga;
    SearchTree izq, der;
}
```

Ejercicio 17.7

En matemática, un **conjunto** es una colección de elementos que no contiene duplicados. La interfaz de `java.util.Set` está destinada a modelar un conjunto matemático. Los métodos que requiere son `add`, `contains`, `containsAll`, `remove`, `size`, y `iterator`⁴.

Escribir una clase llamada `TreeSet`⁵ que sea una extensión de `SearchTree` y que implemente `Set`. Para hacerlo fácil, se puede asumir que `null` no aparece como carga de ningún nodo del árbol, ni como parámetro de ninguno de los métodos.

Ejercicio 17.8

Escribir un método llamado `union` que toma dos `Set` como parámetros y devuelve un nuevo `TreeSet` que contiene todos los elementos que aparecen en alguno de los conjuntos.

Podés agregar este método en tu implementación de `TreeSet`, o crear una nueva clase que extiende `java.util.TreeSet` y que incluya `union`.

Ejercicio 17.9

Escriba un método llamado `interseccion` que toma dos `Set` como parámetros y devuelve un nuevo `TreeSet` que contiene todos los elementos que aparecen en ambos conjuntos.

`union` e `interseccion` son genéricas en el sentido de que los parámetros pueden ser de cualquier tipo en la metaclass `Set`. Los dos parámetros ni siquiera tienen que ser del mismo tipo.

Ejercicio 17.10

Una de las razones por la cual la interface `Comparable` es útil, es que permite a un tipo de objeto especificar cualquier orden que sea apropiado. Para tipos como `Integer` y `Double`, el orden adecuado es obvio, pero hay un montón de ejemplos en los que el orden depende de los objetos representados. En el golf, por ejemplo,

4. N.d.T.: agregar, contiene, contieneTodos, quitar, tamaño e iterador respectivamente.

5. N.d.T.: “Conjunto sobre árbol”.

una puntuación baja es mejor que una puntuación alta, si se comparan dos objetos `Golfista`, el que tiene la puntuación más baja gana.

- a. Escribir una definición de una clase `Golfista` que contenga un nombre y una puntuación entera como variables de instancia. La clase debe implementar `Comparable` y proporcionar un método `compareTo` que da mayor prioridad a la puntuación más baja.
- b. Escribir un programa que lea un archivo que contiene los nombres y los resultados de un conjunto de jugadores de golf. Se deben crear objetos de tipo `Golfista`, ponerlos en una Cola de Prioridad y luego tomarlos e imprimirlos. Deben aparecer en orden descendente de prioridad, que es en orden ascendente por puntuación.

Tiger Woods	61
Hal Sutton	69
Phil Mickelson	72
Allen Downey	158

SUGERENCIA: Consultar el Apéndice C.3 para ver un código que lee las líneas de un archivo.

Ejercicio 17.11

Escribir una implementación de Pila usando un Vector. Pensá acerca de si es mejor poner los nuevos elementos en el inicio o al final del vector.

Capítulo 18

Heap

18.1 Implementación de árbol sobre arreglo

¿Qué quiere decir “implementar” un árbol? Hasta ahora sólo hemos visto una implementación de árbol, una estructura de datos enlazada, similar a la lista enlazada. Pero hay otras estructuras que querríamos identificar como árboles. Cualquier cosa que pueda efectuar el conjunto básico de operaciones de árbol, debe ser reconocido como un árbol.

De modo que, ¿cuáles son las operaciones de árbol? En otras palabras, ¿cómo definimos el TAD Árbol?

constructor: Construir un árbol vacío.

dameIzquierdo: Obtener el hijo izquierdo de este nodo.

dameDerecho: Obtener el hijo derecho de este nodo.

damePadre: Obtener el nodo padre de este nodo.

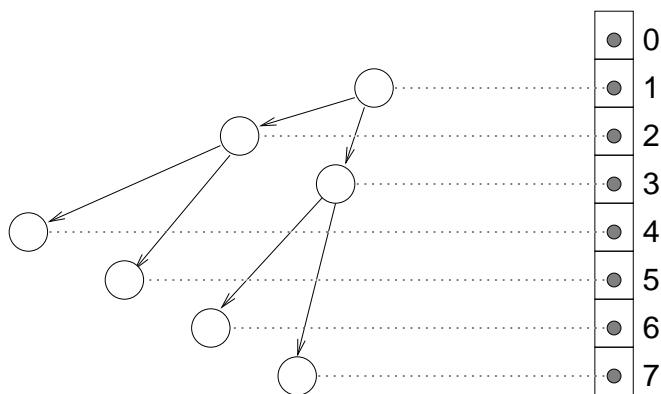
dameCarga: Obtener el objeto carga de este nodo.

ponerCarga: Establecer un objeto carga a este nodo (y crear el nodo, si hace falta).

En la implementación enlazada, el árbol vacío se representa por el valor especial null. **dameIzquierdo** y **dameDerecho** se efectúan accediendo a las variables de instancia del nodo, tal como lo hacen también **dameCarga** y **ponerCarga**. No hemos implementado **damePadre** aún (podrías pensar en cómo hacerlo).

Hay otra implementación de árboles que usa arreglos e índices en lugar de objetos y referencias. Para ver cómo funciona, comenzaremos por mirar una implementación híbrida que usa tanto arreglos como objetos.

Esta figura muestra un árbol como los que hemos visto antes, sin embargo está dibujado en forma oblicua. A la derecha hay un arreglo de referencias, que referencian las cargas de los nodos.



Cada uno de los nodos en el árbol tiene un índice único. Más aún, los índices han sido asignados a nodos de acuerdo a un patrón deliberado, de modo de obtener los siguientes resultados:

1. El hijo izquierdo de un nodo con índice i tiene índice $2i$.
2. El hijo derecho del nodo con índice i tiene índice $2i + 1$.
3. El nodo padre con índice i tiene índice $i/2$ (redondeado hacia abajo).

Usando estas fórmulas, podemos implementar los métodos `damePadre`, `dameIzquierdo` y `dameDerecho` sólo con algunas cuentas; ¡no necesitamos usar referencias para nada!

Dado que no utilizamos referencias, podemos deshacernos de ellas, lo que significa que lo que usamos como nodo de un árbol es ahora la carga y nada más. Eso significa que podemos implementar un árbol como un arreglo de objetos que queremos almacenar; no necesitamos los nodos del árbol. Así es cómo se ve una implementación posible:

```
public class ArbolEnArreglo {
    Object[] arreglo;
    int largo;
```

```

    public ArbolEnArreglo() {
        arreglo = new Object [128];
    }
}

```

No hay sorpresas hasta ahora. La única variable de instancia es un arreglo de Objects que contiene las cargas del árbol. El constructor inicializa el arreglo con una capacidad inicial arbitraria; el resultado es un árbol vacío.

Aquí está la implementación más simple de `dameCarga` y `ponerCarga`.

```

    public Object dameCarga(int i) {
        return arreglo[i];
    }

    public void ponerCarga(int i, Object obj) {
        arreglo[i] = obj;
    }

```

Estos métodos no hacen ninguna verificación, así que si el parámetro está mal puede haber una excepción del tipo `ArrayIndexOutOfBoundsException`.

La implementación de `dameIzquierdo`, `dameDerecho` y `damePadre` es sólo aritmética:

```

    public int dameIzquierdo(int i) { return 2*i; }
    public int dameDerecho (int i) { return 2*i + 1; }
    public int damePadre(int i)   { return i/2; }

```

Finalmente estamos listos para construir un árbol. En otra clase (el cliente), escribiremos

```

ArbolEnArreglo arbol = new ArbolEnArreglo();
arbol.ponerCarga(1, "carga de la raíz");

```

El constructor construye un árbol vacío. Llamar a `ponerCarga` pone la cadena "carga de la raíz" en el nodo raíz. Para añadir hijos al nodo raíz:

```

arbol.ponerCarga(arbol.dameIzquierdo(1), "carga izquierda");
arbol.ponerCarga(arbol.dameDerecho(1), "carga derecha");

```

En la clase para el árbol podríamos proveer un método que imprima los contenidos del árbol en *preorder*.


```

public void imprimir(int i) {
    Object carga = dameCarga(i);
    if (carga== null) return;
    System.out.println (carga);
    imprimir(dameIzquierdo(i));
    imprimir(dameDerecho(i));
}

```

Para llamar a este método, tenemos que pasar el índice de la raíz como parámetro.

```
arbol.imprimir(1);
```

La salida es

```

carga de la raíz
carga izquierda
carga derecha

```

Esta implementación provee las operaciones básicas que definen un árbol. Como ya he mencionado, la implementación enlazada de un árbol provee las mismas operaciones, pero la sintaxis es distinta.

En cierto sentido, la implementación de arreglo es un poco fea. Por un lado, asumimos que una carga null indica que un nodo no existe, pero eso implica que no podemos poner un objeto null en el árbol como carga.

Otro problema es que los subárboles no están representados como objetos; están representados por índices en el arreglo. Para pasar un nodo de un árbol como parámetro, tenemos que pasar una referencia al objeto árbol y un índice del arreglo.

Finalmente, algunas operaciones que son fáciles en la implementación enlazada, como reemplazar un subárbol completo, son más difíciles en la implementación de arreglo.

Por el otro lado, esta implementación ahorra espacio, dado que no hay enlaces entre los nodos, y hay varias operaciones que son más fáciles y más rápidas en la implementación de arreglo. Resulta que estas operaciones son justamente aquellas que queremos para implementar un Heap.

Un Heap es una implementación del TAD Cola de Prioridad que está basada en la implementación de arreglo de un Árbol. Resulta ser más eficiente que otras implementaciones que hemos visto.

Para probar esta afirmación, procederemos en pasos. Primero, tenemos que desarrollar formas de comparar la eficiencia de varias implementaciones. A continuación, veremos qué operaciones efectúa el Heap. Finalmente, compararemos la implementación de la Cola de Prioridad en un Heap con las otras (arreglos y listas) y veremos por qué el Heap es particularmente eficiente.

18.2 Análisis de eficiencia

Cuando comparamos algoritmos, quisiéramos tener una manera de decidir cuál es más rápido, o requiere menos memoria, o usa menos de algún otro recurso. Es difícil responder esas preguntas en detalle, porque el tiempo y la memoria usada por un algoritmo dependen de la implementación del algoritmo, el problema en particular que se resuelve, y el hardware en el que corre el programa.

El objetivo de esta sección es desarrollar una forma de hablar acerca de eficiencia que sea independiente de todas esas cosas, y sólo dependa del algoritmo en sí mismo. Para empezar, comenzaremos por concentrarnos en el tiempo; luego hablaremos de otros recursos. Nuestras decisiones se guían por una serie de restricciones:

1. Primero, la eficiencia de un algoritmo depende del hardware en el cual corre, con lo que por lo general no vamos a hablar de tiempo de ejecución en términos absolutos como segundos. En su lugar, vamos a contar el número de operaciones abstractas que efectúa un algoritmo.
2. Segundo, la eficiencia depende generalmente del problema particular que intentamos resolver – algunos problemas son más fáciles que otros. Para comparar algoritmos, generalmente nos concentramos, o bien en el escenario del peor caso, o en un caso promedio (o común).
3. Tercero, la eficiencia depende del tamaño del problema (generalmente, pero no siempre, el número de elementos en una colección). Abordamos este problema de forma explícita expresando el tiempo de ejecución como función del tamaño del problema.
4. Finalmente, la eficiencia depende de los detalles de implementación, como el costo extra de reservar memoria para los objetos o el costo asociado a la llamada de métodos. Generalmente ignoramos estos detalles porque no afectan la tasa con la que crece el número de operaciones abstractas cuando aumenta el tamaño del problema.

Para hacer este proceso más concreto, considerará dos algoritmos que ya vimos para ordenar un arreglo de enteros. El primero es el **ordenamiento por selección**, que vimos en la Sección 12.3. Este es el pseudocódigo que usamos allí.

```

ordenamientoPorSeleccion (arreglo) {
    for (int i=0; i<arreglo.length; i++) {
        // buscar el mínimo elemento entre los elementos
        //     que están en i o a su derecha
        // intercambiar ese elemento con el elemento en i
    }
}

```

Para efectuar las operaciones especificadas en el pseudocódigo, escribimos métodos auxiliares llamados `buscarMinimo` e `intercambiar`. En pseudocódigo `buscarMinimo` tiene la siguiente forma.

```

// buscar el índice del mínimo elemento entre
// i y el final del arreglo

buscarMinimo(arreglo, i) {
    // minimo contiene el índice del mínimo elemento
    //     encontrado hasta el momento
    minimo = i;
    for (int j=i+1; j<arreglo.length; j++) {
        // comparar el j-ésimo elemento con el mínimo
        //     encontrado hasta el momento
        // si el j-ésimo es menor, reemplazar mínimo con j
    }
    return minimo;
}

```

E `intercambiar` tiene la siguiente forma:

```

intercambiar(i, j) {
    // guardar una referencia al i-ésimo elemento en temp
    // hacer que el i-ésimo elemento en el arreglo
    //     referencie al j-ésimo elemento
    // hacer que el j-ésimo elemento referencie a temp
}

```

Para analizar la eficiencia de este algoritmo, el primer paso es decidir qué operaciones contar. Obviamente, el programa hace muchas cosas: incrementa `i`, lo compara con el largo del mazo, busca el mayor elemento del arreglo, etc. No es obvio qué sería correcto contar.

Resulta que una buena decisión es tomar el número de veces que comparamos dos elementos. Muchas otras elecciones nos hubieran llevado al mismo resultado, pero esto es fácil de hacer y veremos que nos permite comparar algoritmos de ordenamiento de una forma más fácil.

El próximo paso es definir el “tamaño del problema”. En este caso es natural elegir el tamaño del arreglo, al que llamaremos n .

Finalmente, querríamos derivar una expresión que nos diga cuántas operaciones abstractas (en este caso, comparaciones) tendríamos que hacer, en función de n .

Comenzamos por analizar los métodos auxiliares. intercambiar copia varias referencias, pero no hace ninguna comparación, por lo que ignoraremos el tiempo utilizado en efectuar intercambios. buscarMinimo comienza en i y atraviesa el arreglo, comparando cada item con `minimo`. El número de elementos que miramos es $n - i$, con lo que el número total de comparaciones es $n - i - 1$.

Luego, consideramos cuántas veces se llama `buscarMinimo` y cuál es el valor de i cada vez. La última vez que es llamada, i vale $n - 2$ por lo que el número de comparaciones es 1. La iteración anterior efectúa 2 comparaciones y así siguiendo. En la primera iteración i vale 0 de modo que el número de comparaciones es $n - 1$.

Por lo que el número de comparaciones es $1 + 2 + \dots + n - 1$. Esta suma es igual a $n^2/2 - n/2$. Para describir este algoritmo, típicamente ignoramos el término más bajo ($n/2$) y decimos que la cantidad total de procesamiento es proporcional a n^2 . Dado que el término de mayor orden es cuadrático, vamos a decir también que este algoritmo es de **tiempo cuadrático**.





18.3 Análisis de mergesort

En la Sección 12.6 afirmé que mergesort requiere un tiempo proporcional a $n \log n$, pero no expliqué por qué. Ahora lo haré.

Nuevamente, comenzamos mirando el pseudocódigo del algoritmo. Para mergesort es:

```
mergeSort (arreglo) {  
    // buscar el punto medio del arreglo  
    // dividir el arreglo en dos mitades  
    // ordenar recursivamente esas dos mitades  
    // unir ambas mitades y devolver el resultado  
}
```

En cada nivel de recursión, dividimos el arreglo a la mitad, hacemos dos llamadas recursivas, y luego unimos las dos mitades. Gráficamente este proceso se ve así:

	# arreglos	elementos por arreglo	# uniones	comparaciones por unión	trabajo total
	1	n	1	$n-1$	$\sim n$
	2	$n/2$	2	$n/2-1$	$\sim n$
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
	$n/2$	2	$n/2$	$2-1$	$\sim n$
	n	1	0	0	

Cada línea en el diagrama es un nivel de recursión. Arriba, un arreglo entero se divide en dos mitades. Abajo, n arreglos con un elemento se unen para formar $n/2$ arreglos con 2 elementos cada uno.

Las primeras dos columnas de la tabla muestran el número de arreglos en cada nivel y el número de elementos en cada arreglo. La tercera columna muestra el número de uniones que se efectúan en cada nivel de recursión. La siguiente columna es la que requiere más concentración: muestra el número de comparaciones que efectúa cada unión.

Si mirás el pseudocódigo (o tu implementación) de unir, deberías convencerte de que el peor caso toma $m - 1$ comparaciones, donde m es el número total de elementos a unir.

El siguiente paso es multiplicar el número de uniones en cada nivel por la cantidad de trabajo (comparaciones) que requiere cada unión. El resultado es el trabajo total requerido por cada nivel. En este punto tomamos ventaja de un pequeño truco. Sabemos que, al final, estamos sólo interesados en el término de mayor grado del resultado, de modo que podemos ignorar el término -1 en las comparaciones por unión. Si hacemos eso, la cantidad de trabajo de cada nivel es simplemente n .

A continuación, necesitamos saber el número de niveles en función de n . Bueno, comenzamos con un arreglo de n elementos y lo dividimos a la mitad hasta que llega a tener 1. Eso es lo mismo que comenzar en 1 e ir multiplicando por 2 hasta que llegamos a n . En otras palabras, queremos saber cuántas veces tenemos que multiplicar a 2 por sí mismo hasta conseguir n . La respuesta es que el número de niveles, l , es el logaritmo en base 2 de n .

Finalmente, multiplicamos la cantidad de trabajo por nivel, n por el número de niveles, $\log_2 n$ para obtener $n \log_2 n$, como se esperaba. No hay un nombre fácil para esta forma de función; la mayoría de la gente simplemente dice, “ene log ene”.

Puede no resultar obvio que $n \log_2 n$ es mejor que n^2 , pero para valores grandes de n , lo es. Como ejercicio, escribí un programa que imprima $n \log_2 n$ y n^2 para un rango de valores de n .

18.4 Overhead

El análisis de la eficiencia requiere muchas simplificaciones. Primero, ignoramos la mayoría de las operaciones que el programa efectúa y contamos sólo comparaciones. Luego decidimos considerar sólo la eficiencia en el peor caso. Durante el análisis, nos tomamos la libertad de redondear algunas cosas, y cuando terminamos, descartamos sin demasiada preocupación los términos de menor orden.

Cuando interpretamos los resultados de este análisis, debemos tener en cuenta todas estas simplificaciones. Dado que mergesort es $n \log_2 n$, lo consideramos mejor algoritmo que el ordenamiento por selección, pero eso no quiere decir que mergesort sea *siempre* más rápido. Sólo significa que eventualmente, si ordenamos arreglos cada vez más grandes, mergesort ganará.

Cuánto tiempo toma eso depende de los detalles de implementación, incluyendo el trabajo adicional a las comparaciones que contamos que efectúa cada algoritmo. Este trabajo extra se suele llamar **overhead**. No afecta el análisis de la eficiencia, pero sí afecta el tiempo de ejecución del algoritmo.

Por ejemplo, nuestra implementación de mergesort en realidad reserva subarreglos antes de hacer las llamadas recursivas y luego los deja ser quitados de memoria por el *garbage collector* luego de ser unidas. Mirando nuevamente el diagrama de mergesort, podemos ver que la cantidad total de espacio que se reserva es proporcional a $n \log_2 n$, y que el número total de objetos que se reserva es alrededor de $2n$. Toda esa reserva de memoria toma tiempo.

Aun así, es generalmente cierto 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 buen algoritmo es mejor y para valores pequeños de n no importa porque ambos algoritmos son suficientemente buenos.

Como ejercicio, escribí un programa que, para un rango de valores de n , imprima $1000 n \log_2 n$ y n^2 . ¿Para qué valor de n son iguales?

18.5 Implementaciones de la Cola de Prioridad

En el Capítulo 16 hemos visto una implementación de Cola de Prioridad basada en un arreglo. Los elementos en el arreglo están desordenados, de modo que es fácil agregar un nuevo elemento (al final), pero es más complicado quitar un elemento, porque tenemos que buscar aquel con la máxima prioridad.

Una alternativa es una implementación basada en una lista enlazada. En este caso cuando agregamos un nuevo elemento recorremos la lista

y ponemos el nuevo elemento en el lugar correcto. Esta implementación aprovecha una propiedad de las listas, por la cual es sencillo agregar un nuevo nodo en el medio de otros dos. De manera similar, quitar un elemento con la máxima prioridad es fácil, ya que a éste lo conservamos como primer elemento de la lista.

El análisis de eficiencia para estas operaciones es trivial. Agregar un elemento al final de un arreglo o quitar un nodo del principio de una lista toma la misma cantidad de tiempo sin importar la cantidad de elementos. De modo que ambas operaciones son de tiempo constante.

Cuando recorremos un arreglo o una lista, efectuando una operación de tiempo constante en cada elemento, el tiempo de ejecución es proporcional al número de elementos. De modo que quitar un elemento de un arreglo y agregar algo a una lista (ordenada) son ambas de tiempo lineal.

¿De modo que cuánto tiempo toma agregar y quitar n elementos de una Cola de Prioridad? Para la implementación de arreglo, agregar n elementos toma tiempo proporcional a n , pero quitarlos toma más tiempo. La primera operación de quitado tiene que recorrer todos los n elementos; la segunda debe recorrer $n - 1$, y así, hasta la última, que sólo tiene que mirar 1 elemento. Por lo tanto, el tiempo total es $1 + 2 + \dots + n$, lo cual es $n^2/2 + n/2$. Con lo que el tiempo total para agregar y quitar los elementos es la suma de la función lineal y la cuadrática, que sigue siendo una cuadrática.

El análisis de la implementación de lista es similar. La primera vez que se agrega, no requiere ningún recorrer nada, pero luego, tenemos que recorrer al menos parte de la lista cada vez que agregamos un nuevo elemento. En general no sabemos cuánto de la lista tenemos que recorrer, dado que depende de los datos y en qué orden se agregan, pero podemos asumir que en promedio tenemos que recorrer la mitad de la lista. Desafortunadamente, aún recorrer la mitad de la lista es una operación lineal.

De modo que, una vez más, agregar y quitar n elementos, toma un tiempo proporcional a n^2 . Por lo tanto, basado en este análisis no podemos decidir qué implementación es mejor; la implementación de arreglo y la de lista son ambas implementaciones de tiempo cuadrático.

Si implementamos una Cola de Prioridad usando un Heap, podemos efectuar tanto las operaciones de agregado como de quitado en un tiempo proporcional a $\log n$. Por lo que el tiempo total para agregar y quitar los n elementos es $n \log n$, lo cual es mejor que n^2 . Esa es la razón por la cual, al principio del capítulo, dije que un Heap es particularmente eficiente para la implementación de una Cola de Prioridad.

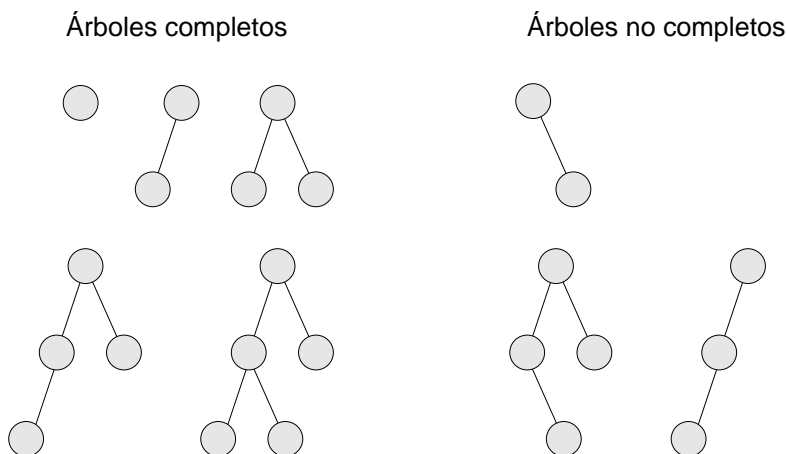
18.6 Definición de un Heap

Un Heap es un tipo particular de árbol. Tiene dos propiedades que no son en general ciertas para otros árboles:

completitud: El árbol es completo, lo cual quiere decir que los nodos se agregan de arriba hacia abajo, de izquierda a derecha, sin dejar ningún espacio.

propiedad de Heap: El elemento en el árbol con mayor prioridad está en la raíz del árbol, y lo mismo es cierto para todo subárbol.

Ambas propiedades requieren un poco de explicación. Esta figura muestra el número de árboles que son considerados completos o no completos.



Un árbol vacío es también considerado completo. Podemos definir a la completitud más rigurosamente comparando la altura de los subárboles. Recordá que la **altura** de un árbol es el número de niveles.

Comenzando en la raíz, si el árbol es completo, entonces la altura del subárbol izquierdo y la altura del subárbol derecho deberían ser iguales, o el árbol izquierdo más alto por una diferencia de uno. En cualquier otro caso, el árbol no puede ser completo. Más aún, si el árbol es completo, entonces la relación de altura entre los subárboles tiene que ser cierta para todo nodo en el árbol.

La **propiedad de heap** es similarmente recursiva. Para que un árbol sea un heap, el valor más grande en el árbol tiene que estar en la raíz, y lo

mismo tiene que valer para cada subárbol. Como otro ejercicio, escribí un método que verifique si un árbol tiene la propiedad de heap.

Ejercicio 18.1

Escribí un método que toma un Árbol como parámetro y verifica si es completo.

PISTA: Podés usar el método `altura` del Ejercicio 17.4.

Ejercicio 18.2

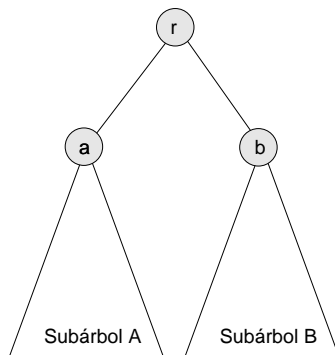
Escribí un método que toma un Árbol como parámetro y verifica si tiene la propiedad de Heap.

18.7 quitar en Heap

Puede parecer raro que vamos a quitar cosas de un Heap antes de agregar ninguna, pero creo que el quitado es más fácil de explicar.

En una primera mirada, podríamos pensar que quitar un elemento de un Heap es una operación de tiempo constante, dado que el elemento con la máxima prioridad está siempre en la raíz. El problema es que una vez que quitamos el nodo raíz, nos quedamos con algo que no es más un Heap. Antes de poder devolver el resultado, debemos restaurar la propiedad de Heap. Llamamos a esta operación `reheapify`.

La situación se muestra en la siguiente figura:



El nodo raíz tiene la prioridad `r` y dos subárboles, A y B. El valor en la raíz del Subárbol A es `a` y el valor en la raíz del Subárbol B es `b`.

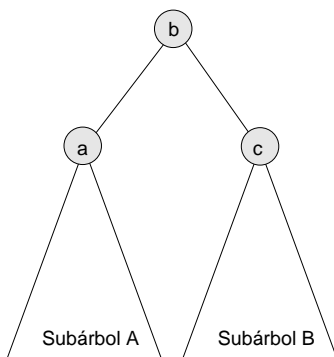
Asumimos que antes de quitar `r` del árbol, el árbol es un Heap. Eso implica que `r` es el valor más grande en el Heap y que `a` y `b` son los valores más grandes en sus respectivos subárboles.

Una vez que quitamos r , tenemos que hacer del árbol resultante un Heap nuevamente. En otras palabras, tenemos que asegurarnos de que tenga las propiedades de completitud y de Heap.

La mejor forma de asegurarnos completitud es quitando el nodo inferior derecho del árbol, al que llamaremos c y poner su valor en la raíz. En una implementación enlazada de árbol, habríamos tenido que recorrer el árbol para encontrar este nodo, pero en una implementación de arreglo, podemos encontrarlo en tiempo constante porque es siempre el último elemento (no nulo) del arreglo.

Desde luego, lo más probable es que este último valor no sea el más grande, de modo que ponerlo en la raíz rompería la propiedad de Heap. Afortunadamente esto es fácil de recuperar. Sabemos que el valor más grande es o bien a o b . Por lo tanto, podemos seleccionar al que sea el más grande de los dos e intercambiarlo con el valor de la raíz.

Arbitrariamente, digamos que b es el más grande. Dado que sabemos que es el valor más grande del Heap, podemos ponerlo en la raíz y poner c en la raíz del Subárbol B. Ahora la situación se ve así:



De nuevo, c es el valor que copiamos del último elemento en el arreglo y b es el más grande del Heap. Dado que no cambiamos el Subárbol A, sabemos que es todavía un Heap. El único problema es que no sabemos si el Subárbol B es un Heap, dado que pusimos un valor (probablemente bajo) en su raíz.

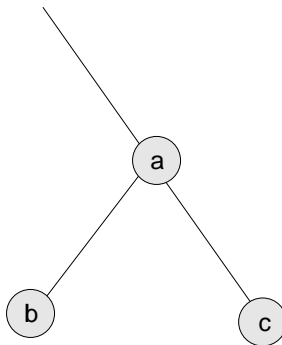
¿No sería lindo tener un método que pudiera hacer un `reheapify` –es decir, devolver la propiedad de Heap– al Subárbol B? Un momento... ¡lo tenemos!

18.8 agregar en Heap

Agregar un nuevo elemento en un Heap es una operación similar, excepto que en lugar de empujar un valor desde arriba hacia abajo, lo empujamos de abajo hacia arriba.

Nuevamente, para garantizar completitud, agregamos el nuevo elemento lo más abajo y a la derecha posible, lo cual es la próxima posición disponible en el arreglo.

Luego, para devolver la propiedad de Heap, comparamos el nuevo valor con sus vecinos. La situación se ve así:



El nuevo valor es c. Podemos recuperar la propiedad de Heap de este subárbol comparando c con a. Si c es más chico, entonces la propiedad de Heap se satisface. Si c es más grande, entonces intercambiar c y a. El intercambio satisface la propiedad de heap porque sabemos que c debe ser también más grande que b, porque $c > a$ y $a > b$.

Ahora que el subárbol está “reheapificado”, podemos seguir repitiendo esta operación hacia arriba en el árbol hasta que lleguemos a la raíz.

18.9 Eficiencia de los Heaps

Tanto para agregar como para quitar, efectuamos una operación de tiempo constante para hacer el agregado o quitado del nodo, pero luego tenemos que recuperar la propiedad de Heap del árbol. En un caso comenzamos en la raíz y empujamos hacia abajo, comparando elementos y luego recursivamente haciendo un reheapify de los subárboles. En el otro caso comenzamos desde una hoja y vamos moviéndonos hacia arriba, nuevamente comparando elementos en cada nivel del árbol.

Como siempre, hay varias operaciones que podríamos querer contar, como comparaciones e intercambios. Cualquier elección funcionaría; el problema real es el número de niveles que tenemos que examinar y qué cantidad de trabajo efectuamos en cada nivel. En ambos casos vamos examinando niveles del árbol hasta que recuperamos la propiedad de Heap, lo que quiere decir que podríamos visitar sólo uno, o en el peor caso podríamos tener que visitarlos a todos. Consideremos el peor caso.

En cada nivel, efectuamos sólo operaciones de tiempo constante como comparaciones e intercambios. De modo que la cantidad total de trabajo es proporcional al número de niveles del árbol – es decir, la altura.

De modo que podríamos decir que estas operaciones son lineales con respecto a la altura del árbol, pero el “tamaño del problema” que nos interesa no es la altura, sino la cantidad de elementos en el Heap.

Como función de n , la altura del árbol es $\log_2 n$. Esto no es cierto para todos los árboles, pero siempre es cierto para árboles completos. Para ver por qué, pensá en el número de nodos en cada nivel del árbol. El primer nivel contiene 1, el segundo contiene 2, el tercero contiene 4, y así siguiendo. El i -ésimo nivel contiene 2^i nodos, y el número total en todos los niveles inferiores al i es $2^i - 1$. En otras palabras, $2^h = n$, lo que implica que $h = \log_2 n$.

Por lo tanto, tanto la operación agregar como quitar toman tiempo **logarítmico**. Agregar y quitar n elementos toma un tiempo proporcional a $n \log_2 n$.

18.10 Heapsort

El resultado de la sección anterior sugiere otro algoritmo de ordenamiento. Dados n elementos, los agregamos a un Heap y luego los quitamos. Dado el comportamiento del Heap, vuelven en orden. Ya hemos mostrado que este algoritmo, el cual se llama **heapsort**, toma tiempo proporcional a $n \log_2 n$, lo cual es mejor que el ordenamiento por selección y lo mismo que mergesort.

A medida que el valor de n se vuelve grande, esperamos que heapsort sea más rápido que el ordenamiento por selección, pero el análisis de eficiencia no nos permite saber si será más rápido que mergesort. Diríamos que ambos algoritmos tienen el mismo **orden de crecimiento** porque sus tiempos de ejecución crecen con funciones que tienen la misma forma. Otra forma de decir esto es que ambos pertenecen a la misma **clase de complejidad**.

Las clases de complejidad se escriben a veces en “notación O-grande”. Por ejemplo, $\mathcal{O}(n^2)$, pronunciado “o de ene cuadrado” es el conjunto de todas las funciones que crecen no más rápido que n^2 para valores gran-

des 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 de creciente son:

$\mathcal{O}(1)$	tiempo constante
$\mathcal{O}(\log n)$	logarítmico
$\mathcal{O}(n)$	lineal
$\mathcal{O}(n \log n)$	“ene log ene”
$\mathcal{O}(n^2)$	cuadrático
$\mathcal{O}(2^n)$	exponencial

Hasta ahora ninguno de los algoritmos que vimos es **exponencial**. Para valores grandes de n , estos algoritmos se vuelven rápidamente poco viables. No obstante, la expresión “crecimiento exponencial” aparece frecuentemente en incluso lenguaje no técnico. Frecuentemente se usa mal, de modo que quería incluir su significado técnico.

La gente usa “exponencial” para describir cualquier curva que aumenta y se va acelerando (es decir, que tiene una inclinación y curvatura positiva). Por supuesto, hay muchas curvas que encajan en esta descripción, incluyendo a las funciones cuadráticas (y polinomios de mayores órdenes) e incluso funciones tan poco dramáticas como $n \log n$. La mayoría de estas curvas no tienen el (a menudo perjudicial) comportamiento de las exponenciales.

18.11 Glosario

ordenamiento por selección: Algoritmo simple de ordenamiento visto en la Sección 12.3.

mergesort: Mejor algoritmo de ordenamiento de la Sección 12.6.

heapsort: Otro algoritmo de ordenamiento.

clase de complejidad: Conjunto de algoritmos cuya eficiencia (generalmente, tiempo de ejecución) tienen el mismo orden de crecimiento.

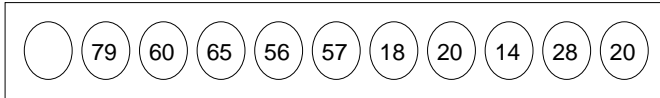
orden de crecimiento: Conjunto de funciones que tiene el mismo término principal, y por lo tanto el mismo comportamiento cualitativo para grandes valores de n .

overhead: Tiempo adicional o recursos consumidos por un programa para efectuar otras operaciones además de las incluidas en el análisis de eficiencia.

18.12 Ejercicios

Ejercicio 18.3

- a. Dibujar el Heap representado por el siguiente arreglo:



- b. Mostrar cómo se vería el arreglo luego de que el valor 68 se agregue al Heap.

Ejercicio 18.4

Asumir que hay n elementos en un Heap. Para encontrar el valor de la mediana de los elementos, podríamos quitar $n/2 - 1$ elementos y luego devolver el valor del $n/2$ -ésimo elemento. Luego, deberíamos tener que poner $n/2$ elementos devuelta en el Heap. ¿Cuál sería el orden de crecimiento de este algoritmo?

Ejercicio 18.5

¿Cuántas veces se ejecutará el siguiente ciclo? Expresá tu respuesta en función de n :

```
while (n > 1) {  
    n = n / 2;  
}
```

Ejercicio 18.6

¿Cuántas llamadas recursivas va a hacer sipo? Expresá tu respuesta en función de x o n o ambos.

```
public static double sipo(double x, int n) {  
    if (n == 0) return 1.0;  
    return x * sipo(x, n-1);  
}
```

Ejercicio 18.7

Escribir una implementación de Heap basada en la implementación de arreglo del árbol. El tiempo de ejecución de agregar y quitar debe ser proporcional a $\log n$, donde n es la cantidad de elementos en el Heap.

Capítulo 19

Maps

19.1 Arreglos, Vectores y Maps

Los arreglos son estructuras de datos usualmente muy útiles, pero tienen dos limitaciones importantes:

- El tamaño del arreglo no depende de la cantidad de elementos en el mismo. Si el arreglo es muy grande, malgasta espacio. Si es muy chico puede llevarnos a causar errores o a tener que escribir código para redimensionarlo.
- Aunque el arreglo puede contener elementos de cualquier tipo, los índices del mismo tienen que ser enteros. No podemos, por ejemplo, usar un String para especificar un elemento de un arreglo.

En la Sección 17.9 vimos cómo la clase preincorporada Vector resuelve el primer problema. A medida que el código cliente agrega elementos el Vector se expande automáticamente. También es posible comprimir el Vector para que la capacidad sea la misma que el tamaño actual.

Pero los Vectores no nos ayudan con el segundo problema. Los índices aún son enteros. Aquí es donde el TAD Map entra en juego. Un Map es similar a un Vector; la diferencia es que puede usar cualquier tipo de objeto como índice. A estos índices generalizados se los llama **claves**.

Así como se usa un índice para acceder a un valor en un vector, se usa una clave para acceder a un valor en un Map. Cada clave se asocia a un valor, y por este motivo los Maps son usualmente llamados **arreglos asociativos**. A la asociación de una clave en particular con un valor en particular se la llama **entrada**.

Un ejemplo típico de un map es un diccionario, el cual relaciona palabras (las claves) con sus definiciones (los valores). Gracias a este ejemplo,

a los Maps también se los suele llamar Diccionarios. Y sólo para completar el tema, los Maps son también llamados Tablas.

19.2 El TAD Map

Tal como el resto de los TADs que vimos, los Maps están definidos por el conjunto de operaciones que soportan:

constructor: Crea un nuevo map vacío.

guardar: Genera una entrada que asocia un valor con una clave.

obtener: Devuelve el valor correspondiente de una determinada clave.

pertenece: Indica si existe una entrada en el map con la clave dada.

claves: Devuelve un conjunto que contiene todas las claves del map.

19.3 El HashMap preincorporado

La clase `java.util.HashMap` es una implementación del TAD Map que viene con Java. Más tarde en este capítulo vamos a ver por qué se llama `HashMap`. La implementación de Java utiliza las siguientes operaciones para implementar el TAD:

guardar: `put`

obtener: `get`

pertenece: `containsKey`

claves: `keySet`

Para demostrar el uso del `HashMap` vamos a escribir un programa corto que recorre un string y cuenta la cantidad de veces que aparece cada palabra. Vamos a crear una nueva clase llamada `CuentaPalabras` que va a construir el Map y después va a imprimir su contenido. Cada objeto `CuentaPalabras` contiene un `HashMap` como variable de instancia:

```
public class CuentaPalabras {
    HashMap map;

    public CuentaPalabras () {
        map = new HashMap ();
    }
}
```

Los únicos métodos públicos de CuentaPalabras son procesarLinea, que toma un String y agrega sus palabras al map, e imprimir, que imprime los resultados al final. El método procesarLinea separa el String en palabras usando un StringTokenizer y pasa cada palabra a procesarPalabra.

```
public void procesarLinea (String s) {
    StringTokenizer st = new StringTokenizer (s, " ,.");
    while (st.hasMoreTokens()) {
        String palabra = st.nextToken();
        procesarPalabra (palabra.toLowerCase ());
    }
}
```

El trabajo interesante está en procesarPalabra:

```
public void procesarPalabra (String palabra) {
    if (map.containsKey (palabra)) {
        Integer i = (Integer) map.get (palabra);
        Integer j = new Integer (i.intValue() + 1);
        map.put (palabra, j);
    } else {
        map.put (palabra, new Integer (1));
    }
}
```

Si la palabra *está en el map* (containsKey), entonces *obtenemos* (get) su contador actual, lo incrementamos, y *asignamos* (put) el nuevo valor. Si no, simplemente *asignamos* (put) una nueva entrada en el map con el contador en 1.

Para imprimir las entradas del map, necesitamos recorrer las claves del map. Por suerte, el HashMap provee un método llamado keySet que devuelve un objeto de tipo Set (un conjunto) con todas las claves actuales del map, y Set provee un método llamado iterator que devuelve un objeto de tipo Iterator (es decir, un iterador) para recorrer el conjunto. Así es cómo se usa keySet para imprimir el contenido del HashMap:

```
public void imprimir () {
    Set claves = map.keySet();
    Iterator it = claves.iterator();
    while (it.hasNext ()) {
        String clave = (String) it.next ();
        Integer valor = (Integer) map.get (clave);
        System.out.println ("{ " + clave + ", " + valor + " }");
    }
}
```

Cada elemento del iterador es un objeto de tipo `Object`, pero como sabemos que son claves, podemos convertir su tipo a `String`. Cuando obtenemos los valores del map, también son de tipo `Object`, pero sabemos que son contadores, entonces los convertimos a enteros. Finalmente, para contar las palabras de un `String`:

```
CuentaPalabras cp = new CuentaPalabras ();
cp.procesarLinea ("dame fuego, dame dame fuego " +
                 "dame el fuego de tu amor");
cp.imprimir ();
```

La salida es

```
{ dame, 4 }
{ el, 1 }
{ fuego, 3 }
{ amor, 1 }
{ de, 1 }
{ tu, 1 }
```

Los elementos del iterador no están en algún orden en particular. Lo único que se garantiza es que todas las claves del map van a aparecer en él.

19.4 Una implementación usando `Vector`

Una manera fácil de implementar el TAD Map es usando un `Vector` de entradas, donde cada entrada es un objeto que contiene una clave y un valor. Una definición de clase para `Entrada` podría ser algo como esto:

```
class Entrada {
    Object clave, valor;

    public Entrada (Object clave, Object valor) {
        this.clave = clave;
        this.valor = valor;
    }

    public String toString () {
        return "{ " + clave + ", " + valor + " }";
    }
}
```

Así, la implementación de Map sería algo como esto:

```

public class MiMap {
    Vector entradas;

    public Map () {
        entradas = new Vector ();
    }
}

```

Para agregar una nueva entrada en el map, simplemente agregamos una nueva Entrada al Vector:

```

public void put (Object clave, Object valor) {
    entradas.add (new Entrada(clave, valor));
}

```

De esta manera, para buscar una clave en el Map tenemos que recorrer el vector y hallar una Entrada con la clave dada:

```

public Object get (Object clave) {
    Iterator it = entradas.iterator ();
    while (it.hasNext ()) {
        Entrada entrada = (Entrada) it.next ();
        if (clave.equals (entrada.clave)) {
            return entrada.valor;
        }
    }
    return null;
}

```

El modo de recorrer un Vector es el que vimos en la Sección 17.10. Cuando comparamos claves, usamos igualdad de contenido (el método equals) en lugar de usar igualdad superficial (el operador ==). Esto permite que la clase de la clave especifique su propia definición de igualdad. En el ejemplo, las claves son Strings, con lo cual el método get usará el método equals de la clase String.

Para muchas de las clases preincorporadas, el método equals implementa igualdad de contenido. Para algunas clases, sin embargo, no es fácil definir qué significa. Por ejemplo, mirá la documentación de equals para Doubles.

Dado que equals es un método de objeto, esta implementación de get no funciona cuando la clave es null. Para poder soportar claves nulas, necesitamos agregar un caso especial a get o escribir un método de clase que compare claves y maneje parámetros null en forma segura. Pero pasaremos por alto este detalle.

Ahora que tenemos `get`, podemos escribir una versión más completa de `put`. Si hay ya alguna entrada en el `map` con la clave dada, `put` debe actualizarla (darle un nuevo valor), y devolver el valor anterior (o `null` si no había ninguno). Aquí hay una implementación que provee esta característica:

```
public Object put (Object clave, Object valor) {
    Object resultado = get (clave);
    if (resultado == null) {
        Entrada entrada = new Entrada(clave, valor);
        entradas.add (entrada);
    } else {
        actualizar (clave, valor);
    }
    return resultado;
}

private void actualizar (Object clave, Object valor) {
    Iterator it = entradas.iterator ();
    while (it.hasNext ()) {
        Entrada entrada= (Entrada) it.next ();
        if (clave.equals (entrada.clave)) {
            entrada.valor= valor;
            break;
        }
    }
}
```

El método `actualizar` no es parte del TAD `Map`, por lo que se declara `private`. Éste recorre el vector hasta que encuentra la `Entrada` correcta y luego actualiza el campo `valor`. Notá que no modificamos el `Vector` mismo, sino los objetos que contiene.

Los únicos métodos que no hemos implementado son `containsKey` y `keySet`. El método `containsKey` es casi idéntico a `get` excepto que devuelve `true` o `false` en lugar de una referencia a un objeto o `null`.

Ejercicio 19.1

Implementar `keySet` a través de la construcción y devolución de un objeto `TreeSet`. Usá tu propia implementación de `TreeSet` del Ejercicio 17.7 o la implementación preincorporada `java.util.TreeSet`.

19.5 La metaclasses `List`

El paquete `java.util` define una metaclasses llamada `List` que especifica el conjunto de operaciones que una clase tiene que implementar para poder ser considerada (muy abstractamente) una lista. Esto no significa, desde ya, que toda clase que implemente `List` deba ser una lista enlazada.

No sorprende que la clase preincorporada `LinkedList` sea un miembro de la metaclasses `List`. En cambio, sí sorprende que lo sea `Vector`.

Los métodos en la definición de `List` incluyen `add`, `get` e `iterator`. De hecho, todos los métodos de `Vector` que utilizamos para implementar `Map` están definidos en la metaclasses `List`. Esto significa que en lugar de un `Vector`, pudimos haber usado cualquier clase que pertenezca a `List`. En nuestra implementación de `Map` podemos reemplazar `Vector` con `LinkedList`, y ¡el programa sigue funcionando!

Este tipo de generalidad puede ser útil para ajustar la eficiencia de un programa. Podés escribir un programa en términos de una metaclasses como `List` y luego probar el programa con distintas implementaciones para ver cuál de ellas se comporta mejor.

19.6 Implementación de `HashMap`

La razón por la que la implementación preincorporada del TAD `Map` se llama `HashMap` es que usa una implementación particularmente eficiente de un `Map` llamada tabla de hash.

Para poder entender la implementación con tabla de hash, y por qué es considerada eficiente, comenzaremos por analizar la eficiencia de la implementación con `List`.

Mirando la implementación de `put`, vemos que hay dos casos. Si la clave no está ya en el `map`, entonces sólo tenemos que crear una nueva entrada y agregarla(`add`) a la `List`. Ambas son operaciones de tiempo constante.

En el otro caso, debemos recorrer la `List` para encontrar la entrada correspondiente. Esa es una operación lineal. Por la misma razón, `get` y `containsKey` son también lineales.

A pesar de que las operaciones lineales son a menudo suficientemente buenas, podemos mejorarlo. ¡Resulta que hay una forma de implementar el TAD `Map` de modo que tanto `put` como `get` sean operaciones de tiempo constante!

La clave está en comprender que recorrer una lista toma un tiempo proporcional al largo de la lista. Si podemos poner una cota al largo de la lista, entonces podemos poner una cota superior al tiempo de recorrido, y cualquier operación con tiempo de ejecución acotado se considera de tiempo constante.

¿Pero cómo limitamos el largo de las listas sin limitar el número de elementos en el map? Incrementando la cantidad de listas. En lugar de una sola lista larga, vamos a tener muchas listas pequeñas.

Mientras sepamos en qué lista buscar, podemos poner una cota en el tiempo de búsqueda.

19.7 Funciones de Hash

Y es ahí donde entran las funciones de hash. Necesitamos una forma de mirar una clave y saber, sin buscar, en qué lista va a estar. Asumiremos que las listas están en un arreglo (o un Vector) de modo que podemos referirnos a ellas por un índice.

La solución pasa por pensar algún tipo de relación—casi cualquier tipo de relación—entre los valores de las claves y los índices de las listas. Para cada clave posible tiene que haber un único índice, pero puede haber muchas claves que mapean al mismo índice.

Por ejemplo, imaginá un arreglo con 8 listas y un map hecho de claves que son Integer y valores que son String. Puede resultar tentador usar el valor entero (mediante intValue) de los Integer como índices, dado que son del tipo correcto, pero hay una cantidad muy grande de enteros que no caen entre 0 y 7, que son los únicos índices válidos.

El operador módulo provee una forma simple (en términos de código) y eficiente (en términos de tiempo de ejecución) de mapear *todos* los enteros en el rango (0, 7). La expresión

```
clave.intValue() % 8
```

garantiza producir un valor que está en el rango que va de -7 a 7 (incluyendo ambos extremos). Si tomás su valor absoluto (usando `Math.abs`) obtenés un índice válido.

Para otros tipos de datos, podemos jugar de manera similar. Por ejemplo, para convertir un `Character` en integer, podemos utilizar el método `Character.getNumericValue` y para los `Double` hay un `intValue`.

Para Strings podríamos obtener el valor numérico de cada carácter y luego sumarlos, o en su lugar, podríamos usar una **suma desplazada**. Para calcular una suma desplazada, alternás entre sumar nuevos valores al acumulador y desplazando el acumulador a la izquierda. Por “desplazar a la izquierda” quiero decir “multiplicar por una constante”.

Para ver cómo funciona esto, tomá por ejemplo la lista de números 1, 2, 3, 4, 5, 6 y calculá su suma desplazada de la siguiente manera. Primero, inicializá el acumulador a 0. Luego,

1. Multiplicá el acumulador por 10.

2. Sumá el siguiente elemento de la lista al acumulador.
3. Repetir hasta acabar la lista.

Como ejercicio, escribí un método que calcula la suma desplazada de los valores numéricos de los caracteres en un `String` usando un multiplicador de 16.

Para cada tipo, podemos pensar una función que toma valores de ese tipo y genera el valor entero correspondiente. Estas funciones se llaman **funciones de hash** y el valor entero para un objeto se dice que es su **código de hash**.

Hay otra forma en la que podemos generar códigos de hash para objetos Java. Cada objeto Java provee un método llamado `hashCode` que devuelve un entero que corresponde a ese objeto. Para los tipos preincorporados, el método `hashCode` está implementado de modo que si dos objetos contienen los mismos datos (igualdad de contenido), tendrán el mismo código de hash. La documentación de estos métodos explica cuál es la función de hash. Deberías darles una mirada.

Para tipos definidos por el usuario, se deja al implementador proveer la función de hash apropiada. La función de hash por omisión, provista por la clase `Object`, usa a menudo la ubicación del objeto para generar su código de hash, de modo que la noción de igualdad está dada por la igualdad superficial. Normalmente cuando estamos buscando una clave en un `Map`, la igualdad superficial no es lo que queremos.

Más allá de cómo se genera el código de hash, el último paso es usar el operador módulo y el valor absoluto para llevar el código de hash al rango válido de índices.

19.8 Redimensionamiento de un `HashMap`

Revisemos lo anterior. Una tabla de hash consiste en un arreglo (o `Vector`) de `List`, donde cada `List` contiene un pequeño número de entradas. Para añadir una nueva entrada al map, calculamos el código de hash de la nueva clave y agregamos la entrada en la `List` correspondiente.

Para buscar nuevamente la clave, debemos obtener su código de hash y buscarla en la lista correspondiente. Si los largos de las listas están acotados, entonces el tiempo de búsqueda está acotado.

¿Entonces cómo mantenemos cortas a las listas? Bueno, una forma es mantenerlas lo más balanceadas que sea posible, de modo que no haya listas muy largas mientras que otras están vacías. Esto no es fácil de lograr a la perfección—depende de qué tan bien elegimos la función de hash—pero generalmente podemos hacer un buen trabajo.

Incluso con balance perfecto, el largo promedio de las listas crece linealmente con el número de entradas, y debemos poner un freno a eso.

La solución es mantener registro de la cantidad promedio de entradas por lista, lo que se conoce como **factor de carga**. Si el factor de carga crece mucho, debemos redimensionar la tabla de hash.

Para redimensionar, creamos una nueva tabla de hash, generalmente el doble de grande que la original, tomamos todas las entradas de la vieja, volvemos a obtener sus códigos de hash, y las ponemos en la nueva tabla. Usualmente podemos utilizar la misma función de hash; simplemente usamos un valor diferente para calcular el módulo.

19.9 Rendimiento del redimensionado

¿Cuánto tiempo toma redimensionar la tabla de hash? Claramente es lineal en el número de entradas. Eso significa que *la mayoría* del tiempo *put* toma un tiempo constante, pero eventualmente —cuando tenemos que redimensionar— toma tiempo lineal.

A primera vista eso suena mal. ¿Acaso eso no contradice la afirmación de que podemos realizar *put* en tiempo constante? Bueno, francamente, sí. Con un poco de maquillaje podemos arreglarlo.

Ya que algunas operaciones de *put* toman más que otras, averigüemos cuál es el tiempo *promedio* de una operación *put*. El promedio va a ser c , el tiempo constante de un simple *put*, más un término adicional p , el porcentaje de veces que hay que redimensionar, multiplicado por kn , el costo en tiempo que toma hacerlo cuando hace falta.

$$t(n) = c + p \cdot kn$$

No sabemos cuánto valen c y k , pero podemos calcular cuánto vale p . Imaginemos que acabamos de redimensionar la tabla de hash duplicando su tamaño. Si hay n entradas, entonces podemos realizar n adiciones antes de redimensionar otra vez. Por lo que el porcentaje de tiempo que tenemos que redimensionar es $1/n$. Volviendo a la ecuación obtenemos

$$t(n) = c + 1/n \cdot kn = c + k$$

En otras palabras, ¡ $t(n)$ es tiempo constante!

19.10 Glosario

map: Un TAD que define operaciones en una colección de entradas.

entrada: Un elemento en un map que contiene una clave y un valor.

clave: Un índice, de cualquier tipo, usado para buscar valores en un map.

valor: Un elemento, de cualquier tipo, guardado en un map.

diccionario: Otro nombre para un map.

arreglo asociativo: Otro nombre para un diccionario.

tabla de hash: Una implementación particularmente eficiente de un map.

función de hash: Una función que relaciona valores de un cierto tipo con enteros.

código de hash: El valor entero que corresponde a un valor.

suma desplazada: Una simple función de hash usada a menudo para objetos compuestos como Strings.

factor de carga: El número de entradas en una tabla de hash dividida por el número de listas; es decir, el número promedio de entradas por lista.

19.11 Ejercicios

Ejercicio 19.2

- Calculá la suma desplazada de los números 1, 2, 3, 4, 5, 6 usando 10 como multiplicador.
- Calculá la suma desplazada de los números 11, 12, 13, 14, 15, 16 usando 100 como multiplicador.

Ejercicio 19.3

Imaginá que tenés una tabla de hash que contiene 100 listas. Si se te pide obtener (mediante `get`) del valor asociado con una cierta clave, y el código hash para esa clave es 654321, ¿Cuál es el índice de la lista en donde estará la clave (si es que está en el diccionario)?

Ejercicio 19.4

Si hubiera 100 listas y 89 entradas en el diccionario, y la más larga contuviera 3 entradas, y la mediana de las longitudes de las listas fuera 1 y el 19 % de las listas estuvieran vacías, ¿Cuál sería el factor de carga?

Ejercicio 19.5

Supongamos que hay una gran cantidad de personas en una fiesta. Te gustaría saber si cualesquiera dos de ellos cumplen años el mismo día. Primero, diseñás

una nueva clase de objeto Java que represente cumpleaños (por ejemplo, `Cumple`), con dos variables de instancia: `mes`, que es un entero entre 1 y 12, y `día`, que es un entero entre 1 y 31.

Luego, creás un arreglo de objetos `Cumple` que contiene un objeto para cada persona en la fiesta, y conseguís a alguien que te ayude a ingresar todos los cumpleaños.

- a. Escribí un método `equals` para `Cumple` de modo que los cumpleaños con mismo mes y día sean iguales.
- b. Escribí un método llamado `tieneDuplicado` que tome un arreglo de `Cumple` y devuelva `true` si hay dos o más personas con el mismo cumpleaños. Tu algoritmo debería recorrer el arreglo de `Cumple` una sola vez.
- c. Escribí un método llamado `cumpleAleatorios` que tome un entero `n` y devuelva un arreglo con `n` objetos `Cumple`, con fechas aleatorias. Para hacerlo fácil, podés hacer de cuenta que los meses tienen todos 30 días.
- d. Generá 100 arreglos aleatorios de 10 `Cumple` cada uno, y fijate cuántos de ellos contienen duplicados.
- e. Si hubiera 20 personas en la fiesta, ¿cuál es la probabilidad de que dos o más cumplan años el mismo día?

Ejercicio 19.6

Se dice que un Diccionario es **inversible** si cada clave y cada valor aparecen una sola vez. En un `HashMap`, es siempre cierto que cada clave aparece una vez sola, pero es posible que el mismo valor aparezca muchas veces. Por lo tanto, algunos `HashMaps` son inversibles y otros no.

Escribí un método llamado `esInversible` que tome un `HashMap` y devuelva verdadero si el diccionario es inversible y falso en caso contrario.

Ejercicio 19.7

El objetivo de este ejercicio es encontrar las 20 palabras más comunes en este libro. Tipeá el código de este libro que usa el `HashMap` preincorporado para contar la frecuencia de las palabras. Descargá el texto del libro de <https://sourceforge.net/projects/thinkcsjava2esp/files/thinkapjavaesp.txt>.

La versión en texto plano del libro es generada automáticamente por un programa llamado `detex` que intenta quitar todos los comandos de edición, pero deja un surtido de porquerías. Pensá cómo tu programa debería lidiar con los signos de puntuación y cualquier otra cosa extraña que aparezca en el archivo. Escribí un programa para responder las siguientes preguntas:

- a. ¿Cuántas palabras hay en el libro?
- b. ¿Cuántas palabras *diferentes* hay en el libro? A modo comparativo, hay (muy a groso modo) 300.000 palabras en el idioma castellano, de las cuales (muy a groso modo) 40,000 son de uso corriente. ¿Qué porcentaje de este vasto diccionario usé en este libro?

- c. ¿Cuántas veces aparece la palabra “encapsulamiento”?
- d. ¿Cuáles son las 20 palabras más comunes en el libro? PISTA: existe otra estructura de datos que puede ayudar con esta parte del ejercicio.

Ejercicio 19.8

Escribí un método para la clase `ListaEnlazada` que chequee si la lista contiene un ciclo. No deberías asumir que el campo `longitud` es correcto. Tu método debe ser lineal en la cantidad de nodos.

Ejercicio 19.9

Escribí una implementación del TAD Diccionario (como está definida en la Sección 19.2) sobre tabla de hash. Testeala usándola con cualquiera de los programas que escribiste hasta ahora. Pistas:

- a. Comenzá con la implementación sobre `Vector` del libro y asegurate que funciona con los programas existentes.
- b. Modificá la implementación para que use, o bien un arreglo de `ListaEnlazada` o de `Vector` de `ListaEnlazada`, el que prefieras.

Ejercicio 19.10

Escribí una implementación de la interfaz `Set`¹ usando `HashMap`.

Ejercicio 19.11

El paquete `java.util` provee dos implementaciones de la interfaz `Map`, llamadas `HashMap` y `TreeMap`. `HashMap` está basada en una tabla de hash como la descrita en este capítulo. `TreeMap` está basada en árbol red-black, que es similar al árbol de búsqueda del ejercicio 17.6.

Si bien estas implementaciones proveen la misma interfaz, esperamos que tengan diferente performance. A medida que el número de entradas, n , incrementa, esperamos que `add` y `contains` tomen tiempo constante para la implementación con tabla de hash, y tiempo logarítmico para la implementación sobre árbol.

Llevá a cabo un experimento para confirmar (¡o refutar!) estas predicciones de performance. Escribí un programa que agregue n entradas a un `HashMap` o `TreeMap`, luego llamá a `contains` con cada una de las claves. Tomá registro del tiempo de ejecución del programa para un rango de valores n y graficalos en función de n . ¿El comportamiento concuerda con nuestras expectativas de performance?

1. N.d.T: Conjunto.

Capítulo 20

Código Huffman

20.1 Códigos de longitud variable

Si estás familiarizado con el código Morse, sabrás que es un sistema para la codificación de las letras del alfabeto como una serie de puntos y rayas. Por ejemplo, la famosa señal ...---... representa las letras SOS, que comprenden una llamada internacionalmente reconocida para pedir ayuda. Esta tabla muestra el resto de los códigos:

A	.-	N	-. .	1	.----	.	..-.-
B	-...	O	---	2	..---	,	--..--
C	-. .	P	-. .	3	...--	?	..-.-.
D	-..	Q	--. -	4-	(-.-.-
E	.	R	-. .	5)	-.-.-.-
F	..- .	S	...	6	-....	-	-....-
G	--.	T	-	7	--... .	"	..-.-.
H	U	..-	8	---..	_	..-.-.-
I	..	V	...-	9	-----	'	..-.-.-
J	.----	W	-. -	0	-----	:	---... .
K	-.-	X	-..-	/	-..- .	;	-.-.-.
L	.-..	Y	-.-.-	+	..-.-.	\$...-.-.-
M	--	Z	--.. .	=	-.-.-.-		

Notá que algunos códigos son más largos que otros. Por diseño, las letras más comunes tienen los códigos más cortos. Dado que hay un número limitado de códigos cortos, esto significa que las letras menos comunes y los símbolos tienen códigos más largos. Un mensaje típico tendrá más códigos cortos que largos, lo que minimiza la media del tiempo de emisión por letra.

A los códigos de este tipo se los conoce como códigos de longitud variable. En este capítulo, vamos a ver un algoritmo para generar un código de longitud variable llamado **Código Huffman**. No sólo es un algoritmo interesante por sí mismo, sino que también lo convierte en un ejercicio útil porque su implementación utiliza muchas de las estructuras de datos que hemos estado estudiando.

He aquí un resumen de las siguientes secciones:

- En primer lugar, vamos a utilizar una muestra de texto en inglés para generar una tabla de frecuencias. Una tabla de frecuencias es como un histograma, que cuenta el número de veces que cada letra aparece en el texto de la muestra.
- El corazón de un código de Huffman es el árbol de Huffman. Vamos a utilizar la tabla de frecuencias para construir el árbol de Huffman, y luego usar el árbol para codificar y decodificar las secuencias.
- Por último, vamos a recorrer el árbol de Huffman y construir una tabla de códigos, que contiene la secuencia de puntos y rayas para cada letra.

20.2 La tabla de frecuencias

Dado que el objetivo es dar códigos cortos a las cartas comunes, tenemos que saber con qué frecuencia se produce cada letra. En el cuento de Edgar Allan Poe “El escarabajo de oro”, uno de los personajes utiliza las frecuencias de las letras para romper un código. Él explica,

“Ahora bien, en inglés, la letra que más frecuentemente se produce es la ‘e’. Posteriormente, la sucesión sigue así: a o i n d h r s t c u f y g w m l b q p k x z. E, sin embargo, predomina tan notablemente que en una oración de cualquier longitud rara vez se ve, que no sea el carácter predominante”.

Así que nuestra primera misión es ver si Poe lo hizo bien. Para comprobarlo, elegí como muestra el texto de “El escarabajo de oro” en sí, que he descargado de uno de los sitios Web de dominio público.

Ejercicio 20.1

Escribí una clase llamada `TablaFrec` que cuente el número de veces que cada letra aparece en un texto de ejemplo. Descargá el texto de tu cuento favorito, siempre y cuando pertenezca al dominio público, y analizá la frecuencia de las letras.

Me pareció más conveniente para hacer `TablaFrec` que herede de `HashMap`. Luego escribí un método llamado `incrementar` que toma una letra como parámetro y agrega o actualiza una entrada en el `HashMap` para cada letra.

Podés utilizar `keySet` para obtener las entradas del `HashMap`, e imprimir una lista de las letras con sus frecuencias. Por desgracia, no aparecerán en ningún orden en particular. El siguiente ejercicio resuelve el problema.

Ejercicio 20.2

Escribí una clase llamada `Par` que representa un par letra-frecuencia. Los objetos `Par` deben contener una letra y una frecuencia como variables de instancia. `Par` debe implementar `Comparable`, donde el `Par` con la frecuencia más alta gana.

Ahora ordená los pares letra-frecuencia del `HashMap` recorriendo el conjunto de claves, creando objetos `Par`, añadiendo los Pares a un `ColaDePrioridad`, quitando los Pares de la `ColaDePrioridad` e imprimiéndolos en orden decreciente de frecuencia.

¿Qué tan buena fue la suposición de Poe acerca de las letras más frecuentes?
¿Qué ocurre con el español?

20.3 El árbol de Huffman

El siguiente paso es construir el árbol de Huffman. Cada nodo del árbol contiene una letra y su frecuencia, y enlaces a los nodos izquierdo y derecho.

Para construir el árbol de Huffman, empezamos por crear un conjunto de árboles de un solo nodo, uno para cada entrada en la tabla de frecuencias. A partir de ahí construimos el árbol de abajo hacia arriba, empezando por las letras de menor frecuencia e iterativamente uniendo subárboles hasta que tengamos un solo árbol que contenga todas las letras.

Aquí está el algoritmo con más detalle.

1. Para cada entrada en la tabla de frecuencias, crear un árbol de Huffman y agregarlo a una `ColaDePrioridad`. Cuando quitamos un árbol de la `ColaDePrioridad`, obtenemos el que tiene la frecuencia más baja.
2. Quitá dos árboles de la `ColaDePrioridad` y unilos mediante la creación de un nodo padre que referencie los nodos extraídos. La frecuencia del nodo padre es la suma de las frecuencias de los hijos.
3. Si la `ColaDePrioridad` está vacía, terminamos. De lo contrario, poner el nuevo árbol en el `ColaDePrioridad` y volver al paso 2.

Un ejemplo hará esto más claro. Para mantener las cosas manejables, vamos a utilizar un texto de ejemplo que sólo contiene las letras `adenrst`:

Santana estrenará sensatas serenatas entre artesanas en Tene-see Street. Narrará tretas de sastres en terrestres tardes. Tanta sanata rentada tenderá redes entre estandartes de seda. Aten-ta de entendedera, Sandra detesta esas densas enredaderas. Retrata desastres, tratará de entender a Dante. Sentada desea a Reed en entrada.

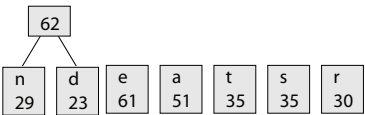
Esta interesante alegoría musical genera la siguiente tabla de frecuencias:

e	61
a	51
s	35
t	35
r	30
n	29
d	23

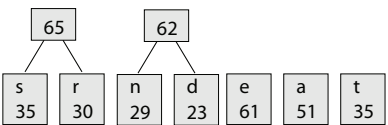
Así que después del paso 1, la ColaDePrioridad se parece a esto:

e	a	t	s	r	n	d
61	51	35	35	30	29	23

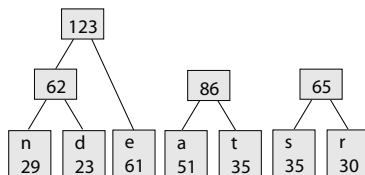
En el paso 2, quitamos los dos árboles con la frecuencia más baja (n y d) y los unimos mediante la creación de un nodo padre con una frecuencia 62. El valor de la letra para los nodos internos es irrelevante, por lo que se omite en las figuras. Cuando ponemos el nuevo árbol en ColaDePrioridad, el resultado es el siguiente:



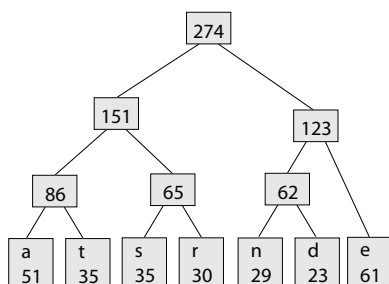
Ahora repetimos el proceso, combinando s y r:



Después de un par de iteraciones, tenemos la siguiente colección de árboles. Por cierto, una colección de árboles se llama **bosque**.



Después de dos repeticiones más, sólo queda un árbol:



Este es el árbol de Huffman para el texto de la muestra. En realidad, este no es el único, porque cada vez que unimos dos árboles, se elige arbitrariamente cuál va a la izquierda y cuál a la derecha, y cuando hay un empate en la ColaDePrioridad, la elección también es arbitraria. Por lo que puede haber muchos árboles posibles para una muestra dada.

Entonces, ¿cómo podemos obtener un código del árbol de Huffman? El código para cada letra está determinado por el camino de la raíz del árbol a la hoja que contiene la letra. Por ejemplo, la ruta de acceso desde la raíz a s es izquierda-derecha-izquierda. Si representamos a la izquierda con . y a la derecha con - (otra elección arbitraria), obtenemos la siguiente tabla de códigos.

e	--
a	...
t	..-
s	.-.
r	.-.
n	-. .
d	-. -

Notá que la 'e', por ser la más frecuente, consigue el código más corto. Las demás reciben todas un código de 3 letras.

Ejercicio 20.3

A mano, deducí un árbol de Huffman para la siguiente tabla de frecuencia:

e	93
s	71
r	57
t	53
n	49
i	44
d	43
o	37

20.4 El método `super`

Una forma de implementar `ArbolHuff` es extender la clase `Par` del Ejercicio 20.2.

```
public class ArbolHuff extends Par implements Comparable {
    ArbolHuff izq, der;

    public ArbolHuff (int frec, String letra,
                     ArbolHuff izq, ArbolHuff der) {
        this.frec = frec;
        this.letra = letra;
        this.izq = izq;
        this.der = der;
    }
}
```

La clase `ArbolHuff` implementa `Comparable` de modo que podamos poner el `ArbolHuff` en una `ColaDePrioridad`. Para implementar la interfaz `Comparable`, temos que proveer un método `compareTo`. Podríamos escribir uno desde cero, pero es más fácil aprovechar la versión de `compareTo` de la clase `Par`.

Lamentablemente, el método preexistente no hace exactamente lo que queremos. Para los pares, le damos prioridad a los elementos de mayor frecuencia. Desde luego, podríamos escribir otra versión de `compareTo`, pero eso sobrescribiría la versión de la clase padre, y nosotros quisiéramos poder llamar al método en la clase padre.

Aparentemente no somos los primeros en encontrarnos este pequeño inconveniente, porque la linda gente que inventó Java nos proveyó de una solución. La palabra clave `super` nos permite llamar a un método que ha sido sobrescrito. Se llama `super` porque las clases padres a veces son llamadas **superclases**¹.

1. N.d.T.: Muy poco frecuente en castellano.

Este es un ejemplo de mi implementación de `ArbolHuff`:

```
public int compareTo (Object obj) {  
    return -super.compareTo (obj);  
}
```

Cuando se llama a `compareTo` en un `ArbolHuff`, éste llama a la versión sobrecrita de `compareTo` y luego devuelve el resultado pasado a negativo, que tiene el efecto de invertir el orden de prioridad.

Cuando una clase hija (llamada también **subclase**) sobrescribe un constructor, puede llamar al constructor del padre usando `super`:

```
public ArbolHuff(int frec, String letra,  
                ArbolHuff izq, ArbolHuff der) {  
    super (frec, letra);  
    this.izq = izq;  
    this.der = der;  
}
```

En este ejemplo, el constructor del padre inicializa `frec` y `letra`, y luego el constructor del hijo inicializa `izq` y `der`.

A pesar de que esta característica es útil, es también propensa a errores. Hay algunas restricciones raras—el constructor del padre debe ser llamado antes de que cualquier otra variable de instancia sea inicializada—y hay algunas sutilezas de las que ni siquiera querés enterarte. En general, este mecanismo es como un botiquín de primeros auxilios. Si te metés en problemas, te puede ayudar. Pero, ¿sabés qué es aún mejor? No meterte en problemas. En este caso, es más simple inicializar las cuatro variables de instancia en el constructor del hijo.

Ejercicio 20.4

Escribí la definición de clase de `ArbolHuff` de esta sección y añadí un método llamado `construir` que toma una `TablaFrec` y devuelve un `ArbolHuff`. Usá el algoritmo de la Sección 20.3.

20.5 Decodificando

Cuando recibimos un mensaje codificado, utilizamos el árbol de Huffman para decodificarlo. Este es el algoritmo:

1. Comenzar en la raíz del `ArbolHuff`.
2. Si el siguiente símbolo es `.`, ir al nodo izquierdo; de lo contrario, ir al derecho.

3. Si estás en una hoja, obtener la letra del nodo y añadirla al resultado. Volver a la raíz.
4. Ir al paso 2.

Considerá el código `..-.-.....`, como ejemplo. Comenzando al inicio del árbol, vamos izquierda - izquierda - derecha y obtenemos la letra t. Luego comenzamos de la raíz nuevamente, vamos izquierda - derecha - derecha, y obtenemos la letra r. Volvemos al inicio, luego derecha - derecha, y obtenemos la letra e. Por último, volvemos al inicio y hacemos derecha - izquierda - izquierda y obtenemos la letra n. Si el código está bien formado, deberíamos estar en una hoja cuando el código termina. En este caso, el mensaje es “tren”.

Ejercicio 20.5

Usá el `ArbolHuff` de ejemplo para decodificar las siguientes palabras:

- a. `.-.-...-.-.-.-.-`
- b. `.-....-.-...-.-`
- c. `..-.-.-...-.-.-.-.-`
- d. `.-.-....-.-...-.-.-.-.-`

Notá que hasta que comiences a decodificar, no podés saber cuántas letras hay, o dónde es que caen los límites de cada letra.

Ejercicio 20.6

Escribí la definición de clase para `Huffman`. El constructor debe tomar un `String` que contiene un texto, y debe construir una tabla de frecuencias y un árbol de `Huffman`.

Escribí un método llamado `decodificar` que toma un `String` que contiene puntos y rayas y que utiliza el árbol de `Huffman` para decodificarlo y devolver el resultado.

Nota: incluso si usás el mismo texto de la Sección 20.2, no vas a obtener necesariamente el mismo `ArbolHuff`, de modo que (probablemente) no vas a poder utilizar tu programa para decodificar los ejemplos en el ejercicio precedente.

20.6 Codificando

En cierto sentido, codificar un mensaje es más difícil que decodificarlo, porque para una letra en particular, podríamos tener que buscar en el árbol el nodo hoja que contiene la letra, y luego descubrir el camino desde la raíz a ese nodo.

Este proceso es mucho más eficiente si recorremos el árbol una sola vez, calculamos todos los códigos, y construimos un `Map` de letras a códigos.

Al momento hemos visto muchas formas de recorrer un árbol, pero esta es inusual porque a medida que nos movemos en el árbol, queremos mantener registro del camino por el que llegamos. Al principio, eso puede parecer difícil, pero hay una forma natural de efectuar este cómputo recursivamente. Para eso necesitamos hacer una observación clave: si el camino de la raíz a un nodo particular es representado por una cadena de puntos y rallas llamada *camino*, entonces el camino al hijo izquierdo de ese nodo es *camino + '-'* y el camino al hijo derecho es *camino + '.'*.

Ejercicio 20.7

- Escribí una definición de clase para *TablaCodigo*, que extiende *HashMap*.
- En la definición de *TablaCodigo*, escribí un método recursivo que se llame *obtenerCodigos* que recorre el *ArbolHuff* en cualquier orden. Cuando llega a un nodo hoja, debe imprimir la letra en el nodo y el código que representa el camino desde la raíz.
- Una vez que *obtenerCodigos* esté funcionando, modificarlo de modo que cuando llegue a un nodo hoja, cree una entrada en el *HashMap*, con la letra como clave y el código como valor.
- Escribí un constructor para *TablaCodigo* que tome un *ArbolHuff* como parámetro y que invoque a *obtenerCodigos* para construir la tabla de códigos.
- En la clase *Huffman*, escribí un método llamado *codificar* que recorra un *String*, busque cada carácter en la tabla de códigos, y devuelva el texto codificado. Probar este método pasándole el resultado a *decodificar* y comprobá si estás obteniendo nuevamente la cadena original.

20.7 Glosario

bosque: Colección de árboles (¡oh!).

superclase: Otro nombre para una clase padre.

subclase: Otro nombre para una clase hija.

super: Palabra clave que puede ser utilizada para llamar a un método sobrescrito de una clase padre.

Apéndice A

Planificación del desarrollo de un programa

Si estás perdiendo mucho tiempo depurando, seguramente es porque no tenés un *plan de desarrollo del programa* efectivo.

Un típico plan de desarrollo de programas erróneo es algo como esto:

1. Escribir un método entero.
2. Escribir algunos métodos más.
3. Intentar compilar el programa.
4. Perder una hora encontrando errores de sintaxis.
5. Perder una hora encontrando errores en tiempo de ejecución.
6. Perder tres horas encontrando errores de semántica.

El problema claramente son los primeros dos pasos. Si escribís más de un método, o incluso un método completo antes de empezar a depurar, probablemente vas a escribir más código del que puedas depurar.

Si te encontrás en esta situación, la *única* solución es quitar código hasta que tengas tu programa funcionando nuevamente, y luego gradualmente reconstruir el programa. Los programadores novatos suelen no querer hacer esto, porque su código, cuidadosamente armado, espreciado para ellos. ¡Para depurar eficientemente, tenés que ser despiadado con tu código! El siguiente es un plan de desarrollo de programas mejor:

1. Empezá con un programa que funcione que haga algo visible, como imprimir algo por pantalla.

2. Agregá cantidades pequeñas de código de a poco, y probá el programa luego de cada cambio.
3. Repetí este procedimiento hasta que el programa haga lo que tiene que hacer.

Luego de cada cambio, el programa debería producir algún efecto visible que muestre el nuevo código. Este enfoque de la programación puede ahorrar mucho tiempo. Ya que sólo se agregan unas pocas líneas en cada paso, es fácil hallar los errores de sintaxis. A su vez, como cada versión del programa produce un resultado visual, estás constantemente testeando tu modelo mental de cómo funciona el programa. Si tu modelo mental está equivocado te vas a enfrentar con el conflicto (y vas a tener la oportunidad de corregirlo) antes de haber escrito un montón de código erróneo.

Un problema de este enfoque es que suele ser difícil imaginarse un camino desde un programa inicial hasta un programa completo y correcto.

Haremos una demostración desarrollando un método llamado `estaEn` que toma una cadena y un carácter, y devuelve un booleano: `true` si el carácter aparece en la cadena y `false` si no.

1. El primer paso es escribir el método más corto posible, tal que compile, ejecute y haga algo visible:

```
public static boolean estaEn (char c, String s) {  
    System.out.println ("estaEn");  
    return false;  
}
```

Obviamente, para probar el método hay que llamarlo. En el `main`, o en algún lugar de un programa que funcione, tenemos que crear un caso de test simple.

Empezaremos con un caso en donde el carácter aparezca en la cadena (así que esperamos que el resultado sea `true`).

```
public static void main (String[] args) {  
    boolean test = estaEn('n', "banana");  
    System.out.println (test);  
}
```

Si todo va de acuerdo al plan, este código compilará, ejecutará e imprimirá la palabra `estaEn` y el valor `false`. Obviamente, la respuesta no es correcta, pero hasta el momento sabemos que el método está siendo llamado y devolviendo un valor.

En mi carrera como programador, he malgastado demasiado tiempo depurando un método, sólo para darme cuenta de que no estaba siendo llamado. Si hubiera usado este plan de desarrollo, esto nunca me hubiese pasado.

2. El siguiente paso es verificar los parámetros que el método está recibiendo.

```
public static boolean estaEn (char c, String s) {  
    System.out.println ("estaEn buscando " + c);  
    System.out.println ("en la cadena " + s);  
    return false;  
}
```

La primera sentencia de impresión nos permite confirmar que el método `estaEn` está buscando la letra correcta. La segunda sentencia confirma que estamos buscando en el lugar indicado.

Ahora la salida se ve así:

```
estaEn buscando n  
en la cadena banana
```

Imprimir los parámetros puede parecer algo tonto, ya que sabemos qué se supone que sean. El objetivo es confirmar que son lo que pensamos que son.

3. Para recorrer la cadena, podemos aprovechar el código de la Sección 7.3. En general, es una gran idea reutilizar fragmentos de código en lugar de reescribirlos desde cero.

```
public static boolean estaEn (char c, String s) {  
    System.out.println ("estaEn buscando " + c);  
    System.out.println ("en la cadena " + s);  
  
    int indice = 0;  
    while (indice < s.length()) {  
        char letra = s.charAt (indice);  
        System.out.println (letra);  
        indice = indice + 1;  
    }  
    return false;  
}
```

Ahora cuando ejecutamos el programa, se imprimen los caracteres de la cadena uno a la vez. Si todo va bien, podemos confirmar que el ciclo examina todas las letras de la cadena.

4. Hasta ahora no hemos pensado mucho en lo que este método va a hacer. En este punto, probablemente necesitemos deducir un algoritmo. El algoritmo más simple es una búsqueda lineal, que recorra la cadena y compare cada elemento con la letra buscada.

Afortunadamente, ya hemos escrito el código que recorre la cadena. Como de costumbre, procedemos agregando sólo unas líneas por vez:

```
public static boolean estaEn (char c, String s) {
    System.out.println ("estaEn buscando " + c);
    System.out.println ("en la cadena " + s);

    int indice = 0;
    while (indice < s.length()) {
        char letra = s.charAt (indice);
        System.out.println (letra);
        if (letra == c) {
            System.out.println ("¡la encontré!");
        }
        indice = indice + 1;
    }
    return false;
}
```

A medida que recorremos la cadena, comparamos cada letra con el carácter buscado. Si lo encontramos, imprimimos algo, para que al ejecutar el nuevo código, se produzca un efecto visible.

5. A esta altura estamos muy cerca de tener un código correcto. El siguiente paso es salir del método si encontramos lo que estábamos buscando:

```
public static boolean estaEn (char c, String s) {
    System.out.println ("estaEn buscando " + c);
    System.out.println ("en la cadena " + s);

    int indice = 0;
    while (indice < s.length()) {
        char letra = s.charAt (indice);
```

```

        System.out.println (letra);
        if (letra == c) {
            System.out.println ("¡la encontré!");
            return true;
        }
        indice = indice + 1;
    }
    return false;
}

```

Si encontramos el carácter buscado, devolvemos true. Si recorremos toda la cadena sin encontrarlo, entonces el valor de retorno correcto es false.

Si ejecutamos el programa en este punto, deberíamos obtener

```

estaEn buscando n
en la cadena banana
b
a
n
¡la encontré!
true

```

6. El siguiente paso es asegurarse que los demás casos de prueba funcionan correctamente. Primero, debemos confirmar que el método devuelve false si el carácter no está en la cadena. Luego deberíamos probar con los típicos valores problemáticos, como la cadena vacía, "", o una cadena con un único carácter.

Como de costumbre, este tipo de testeo puede ayudar a encontrar errores de haberlos, pero no puede asegurarte que el método sea correcto.

7. El anteúltimo paso es quitar o comentar las sentencias de impresión.

```

public static boolean estaEn (char c, String s) {
    int indice = 0;
    while (indice < s.length()) {
        char letra = s.charAt (indice);
        if (letra == c) {
            return true;
        }
        indice = indice + 1;
    }
}

```

```

    }
    return false;
}

```

Comentar las sentencias de impresión es una buena idea si creés que vas a tener que volver a tocar el código de este método más adelante. Aunque si esta es la versión final de este método, y estás convencido que el método es correcto, deberías quitarlas.

Sacar los comentarios te permite ver el código más claramente, lo cual puede ayudarte a detectar cualquier problema restante.

Si hay algo acerca del código que no es obvio deberías agregar comentarios para explicarlo. Hay que resistir la tentación de traducir el código línea por línea. Por ejemplo, nadie necesita esto:

```

// si letra es igual a c, devolver true
if (letra == c) {
    return true;
}

```

Deberías usar comentarios para explicar código que no es obvio, o para advertir sobre condiciones que podrían causar errores, y para documentar cualquier presunción que se haya tomado para el código. También, antes de cada método es una buena idea escribir una descripción resumida de lo que hace el método.

8. El paso final es examinar el código y ver si podés convencerte a vos mismo de que éste es correcto. A esta altura ya sabemos que el método es sintácticamente correcto, ya que compila. Para verificar errores en tiempo de ejecución deberías encontrar cada sentencia que pueda causar un error y deducir las condiciones para que el error aparezca.

En este método, la única sentencia que puede causar un error en tiempo de ejecución es `s.charAt (indice)`. Esta sentencia fallará si `s` es `null` o si el índice está fuera de rango. Dado que obtenemos `s` como parámetro, no podemos estar seguros de que no es `null`; lo único que podemos hacer es verificarlo. En general, es una buena idea que los métodos verifiquen que sus parámetros son válidos. La estructura del ciclo `while` asegura que `indice` está siempre entre 0 y `s.length()-1`. Si verificamos todas las condiciones problemáticas, o demostramos que nunca pueden suceder, entonces podemos probar que este método no causará errores en tiempo de ejecución.

No hemos demostrado aún que el método es semánticamente correcto, pero trabajando incrementalmente, hemos evitado muchos posibles errores. Por ejemplo, ya sabemos que el método está recibiendo los parámetros correctamente y que el ciclo recorre la cadena por completo. También sabemos que está comparando exitosamente los caracteres y que devuelve true si encuentra el carácter buscado. Finalmente, sabemos que si el ciclo termina, el carácter buscado no está en la cadena.

A falta de una demostración formal, eso es lo mejor que podemos hacer.

Apéndice B

Depuración

Hay diferentes tipos de errores que pueden ocurrir en un programa, y es útil distinguir entre ellos a fin de localizarlos con mayor rapidez.

- Los errores en tiempo de compilación son los que produce el compilador y por lo general indican que hay algo malo en la sintaxis del programa. Ejemplo: omitiendo el punto y coma al final de una sentencia.
- Los errores en tiempo de ejecución son los que produce el sistema si algo sale mal mientras se ejecuta el programa. La mayoría de los errores en tiempo de ejecución son excepciones. Ejemplo: un bucle infinito finalmente provoca una excepción de desbordamiento de pila (`StackOverflowException`).
- Los errores semánticos son los problemas que hay con un programa que compila y se ejecuta, pero no hace lo correcto. Ejemplo: una expresión puede no ser evaluada en el orden que esperás, dando un resultado inesperado.

El primer paso en la depuración es averiguar con qué tipo de error te estás enfrentando. Aunque las siguientes secciones se organizan por tipo de error, hay algunas técnicas que se aplican en más de una situación.

B.1 Errores en tiempo de compilación

El mejor tipo de depuración es cuando no tenés que hacerlo, porque estarías evitando cometer errores en el primer lugar. En la sección anterior, te sugerí un plan para el desarrollo de un programa que reduzca al

mínimo el número de errores que podés cometer y lograr que sea fácil encontrarlos cuando los cometás. La clave es comenzar con un programa que funcione y agregar una pequeña cantidad de código a la vez. De esta forma, cuando hay un error, tendrás una idea bastante buena de dónde se encuentra.

Sin embargo, puede ser que te encuentres en alguna de las siguientes situaciones. Para cada una de las situaciones, te hago algunas sugerencias sobre cómo proceder.

El compilador está tirando mensajes de error

Si el compilador tira 100 informes de mensajes de error, eso no significa que hay 100 errores en tu programa. Cuando el compilador encuentra un error, a menudo pierde el control por un tiempo. Intenta recuperarse y empezar de nuevo después del primer error, pero a veces falla, y reporta errores falsos.

En general, sólo el primer mensaje de error es confiable. Te sugiero que sólo corrijas un error a la vez, y luego vuelvas a compilar el programa. Podés encontrar que un punto y coma “soluciona” 100 errores. Por supuesto, si ves varios mensajes de errores legítimos, podés corregir más de un error por intento de compilación.

Recibo un mensaje extraño del compilador y no quiere desaparecer

En primer lugar, leé cuidadosamente el mensaje de error. Está escrito en un lenguaje lacónico, pero a menudo hay un núcleo de información cuidadosamente oculto.

Por lo menos, el mensaje te dirá en qué parte del programa se produjo el problema. En realidad, te informará dónde fue que el compilador se dio cuenta de un problema, que no necesariamente es donde está el error. Utilizá la información que el compilador te da como una guía, pero si no ves un error donde el compilador está señalando, ampliá la búsqueda.

En general, el error estará antes de donde señala el mensaje de error, pero hay casos en los que estará en algún sitio completamente distinto. Por ejemplo, si recibe un mensaje de error en una llamada a un método, el error real puede estar en la definición del método.

Si estás construyendo el programa de forma incremental, debés tener una idea bastante buena de dónde está el error. Será en la última línea que agregaste.

Si estás copiando el código de un libro, comenzá por comparar el código del libro con tu código muy cuidadosamente. Revisá todos los caracteres. Al mismo tiempo, recordá que el libro podría estar equivocado, así que si ves algo que parece un error de sintaxis, eso podría ser el problema.

Si no encontrás el error rápidamente, tomá un respiro y mirá de manera más amplia el programa entero. Ahora es un buen momento para recorrer todo el programa y asegurarte de que tiene una tabulación correcta. No voy a decir que una buena tabulación facilita el encontrar errores de sintaxis, pero seguro que una mala lo hace más difícil.

Ahora, empecemos a examinar el código para los errores de sintaxis común.

1. Comprá que todos los paréntesis y las llaves estén equilibradas y anidadas correctamente. Todas las definiciones de un método deben estar anidadas dentro de la definición de una clase. Todas las sentencias del programa deben estar dentro de la definición de un método.
2. Recordá que las letras mayúsculas no son lo mismo que las minúsculas.
3. Comprá si hay un punto y coma al final de las sentencias (y no un punto y coma después de llaves cerradas).
4. Asegurate de que todas las cadenas en el código se encuentren entre comillas. Asegurate de que utilizás comillas dobles para cadenas y comillas simples para caracteres.
5. Para cada sentencia de asignación, asegurate de que el tipo de dato de la izquierda es el mismo que el tipo de dato de la derecha. Asegurate de que la expresión de la izquierda es un nombre de variable u otra cosa que se puede asignar un valor (como un elemento de un arreglo).
6. Para cada llamada de método, asegurate de que los parámetros que le pasás están en el orden correcto, y tienen el tipo de dato correcto, y que el objeto sobre el que estás llamando el método es del tipo correcto.
7. Si estás llamando un método con resultado, asegurate de que estás haciendo algo con el resultado. Si llamás a un método de tipo void, asegurate de que no estás tratando de hacer algo con el resultado.
8. Si estás llamando a un método de objeto, asegurate de que lo estás llamando en un objeto con el tipo de dato correcto. Si se llama a un método de clase que está fuera de la clase donde se define, asegurate de especificar el nombre de clase.
9. Dentro de un método de objeto podés hacer referencia a las variables de instancia sin la especificación de a qué objeto pertenecen.

Si lo intentás en un método de clase, recibirás un mensaje confuso como “referencia estática a una variable no estática”.

Si nada funciona, pasar a la siguiente sección...

No puedo conseguir que mi programa compile, no importa lo que haga

Si el compilador dice que hay un error y no lo ves, esto puede deberse a que vos y el compilador no están viendo el mismo código. Revisá tu entorno de desarrollo para asegurarte de que el programa que estás editando es el programa que el compilador está compilando. Si no estás seguro, tratá de poner un evidente y deliberado error de sintaxis justo en el comienzo del programa. Ahora compilá de nuevo. Si el compilador no encuentra el nuevo error, probablemente hay algo malo en la manera de configurar el proyecto.

De lo contrario, si examinaste a fondo el código, es el momento de tomar medidas desesperadas. Empezá de nuevo con un programa que puedas compilar y, a continuación, añadir poco a poco tu código de nuevo.

- Hacé una copia del archivo que estás trabajando. Si estás trabajando en `Pepe.java`, hacé una copia llamada `Pepe.java.viejo`.
- Eliminá la mitad del código de `Pepe.java`. Tratá de compilar de nuevo.
 - Si el programa se compila ahora, entonces sabés que el error está en la otra mitad. Volvé a agregar alrededor de la mitad del código que borraste y repetir.
 - Si el programa todavía no se compila, el error debe estar en esta mitad. Eliminar la mitad del código restante y repetir.
- Una vez que hayas identificado y corregido el error, comenzar a agregar el código que fue borrado, un poco a la vez.

Este proceso se llama “depuración por bisección”. Como alternativa, podés comentar trozos de código en lugar de eliminarlos. Sin embargo, para descubrir problemas muy complejos de sintaxis, creo que la eliminación es más fiable—no tenés que preocuparte por la sintaxis y los comentarios, y haciendo el programa más pequeño, hacemos que sea más legible.

Hice lo que el compilador me dijo que hiciera, pero aun así no funcionó

Algunos mensajes del compilador vienen con pedacitos de asesoramiento, como “class Golfista must be declared abstract. It does not define

`int compareTo(java.lang.Object)` from interface `java.lang.Comparable`¹. Parece que el compilador te está diciendo que declares `Golfista` como una clase abstracta, y si estás leyendo este libro, es probable que no sepas lo que es o cómo se hace.

Afortunadamente, el compilador está mal. La solución en este caso es asegurarse de que `Golfista` tenga un método `compareTo` que tome un `Object` como parámetro.

En general, no dejes que el compilador te lleve de la nariz. Los mensajes de error pueden darte evidencia de que algo está mal, pero puede ser engañoso, y sus “consejos” a menudo se equivocan.

B.2 Errores en tiempo de ejecución

Mi programa se cuelga

Si un programa se detiene y parece que no está haciendo nada, decimos que está “colgado”. A menudo esto significa que está atrapado en un ciclo infinito o una recursividad infinita.

- Si hay un ciclo particular que sospeches que puede ser el problema, agregá una sentencia de impresión inmediatamente antes del bucle que dice “entrar en el ciclo” e inmediatamente después otra que dice “salir del ciclo”.

Ejecutá el programa. Si recibís el primer mensaje y no el segundo, tenés un ciclo infinito. Ir a la sección titulada “Ciclo infinito”.

- La mayoría de las veces una recursividad infinita hará que el programa se ejecute un tiempo y luego produce una excepción de desbordamiento de pila (`StackOverflowException`). Si esto ocurre, andá a la sección titulada “recursividad infinita”.

Si no estás recibiendo una excepción de desbordamiento de pila (`StackOverflowException`), pero sospechás que hay un problema con un método iterativo, podés seguir utilizando las técnicas de la sección de recursividad infinita.

- Si ninguna de estas cosas funcionó, empezar a probar otros ciclos y otros métodos recursivos.
- Si ninguna de estas sugerencias te ayuda, entonces es posible que no entiendas el flujo de ejecución de tu programa. Ir a la sección titulada “Flujo de ejecución”.

1. N.d.T.: “clase `Golfista` debe ser declarada como abstract. No está definida `int compareTo (java.lang.Object)` de la interfaz `java.lang.Comparable`”.

Ciclo infinito

Si pensás que tenés un ciclo infinito y pensás que sabés qué ciclo está causando el problema, agregá una sentencia de impresión en el final del ciclo que imprima los valores de las variables en la condición, y el valor de la condición. Por ejemplo,

```
while (x > 0 && y < 0) {  
    // hace algo con x  
    // hace algo con y  
  
    System.out.println ("x: " + x);  
    System.out.println ("y: " + y);  
    System.out.println ("condicion: " + (x > 0 && y < 0));  
}
```

Ahora al ejecutar el programa podrás ver tres líneas producto de cada vez que pasa por el ciclo. La última vez que pasó el ciclo, la condición debe ser false. Si el ciclo sigue adelante, podrás ver los valores de x e y y averiguar por qué no se actualiza correctamente.

Recursividad infinita

La mayoría de las veces una recursión infinita hará que el programa se ejecute por un tiempo y luego producirá una excepción de desbordamiento de pila (`StackOverflowException`).

Si sabés que un método está causando una recursión infinita, comenzá por comprobar, para asegurarte, de que existe un caso base. En otras palabras, debe haber una cierta condición que hará que el método termine sin hacer una llamada recursiva. Si no, entonces necesitás repensar el algoritmo e identificar un caso base.

Si hay un caso base, pero el programa no parece ser que llegue al mismo, agregá una sentencia de impresión al principio del método que muestre los parámetros. Ahora al ejecutar el programa podrás ver en la salida los parámetros con que cada vez se llama al método. Si los parámetros no se acercan hacia el caso de referencia, podrás hacerte alguna idea sobre por qué no.

Flujo de ejecución

Si no estás seguro de cómo el flujo de ejecución se está moviendo a través de tu programa, agregá sentencias de impresión al comienzo de cada método con un mensaje como “entrando al método esPrueba”, donde esPrueba es el nombre del método.

Ahora al ejecutar el programa se imprimirá un rastro de los métodos que se van llamando.

A menudo es útil imprimir los parámetros que cada método recibe cuando se llama. Cuando se ejecuta el programa, compróba si los parámetros son razonables, y si no estás cometiendo uno de los errores clásicos—pasar los parámetros en el orden equivocado.

Cuando ejecuto el programa obtengo una excepción

Si algo va mal durante el tiempo de ejecución, el sistema de tiempo de ejecución de Java imprime un mensaje que incluye el nombre de la excepción, la línea del programa donde se produjo el problema, y una traza de la pila de llamadas. La traza de la pila incluye el método que se está ejecutando actualmente, y luego el método que lo llamó, a continuación el método que invocó *a éste*, y así sucesivamente. En otras palabras, describe la pila de llamadas a métodos que llegaron a donde estás.

El primer paso es examinar el lugar en el programa donde se produjo el error y ver si puede averiguar lo que pasó.

NullPointerException: Excepción de puntero nulo, intentaste acceder a una variable o llamar un método en un objeto null. Tenés que descubrir qué variable es null y luego averiguar cómo llegó a estar así. Recordá que cuando se declara una variable con un tipo de objeto, inicialmente vale null, hasta que se asigna un valor a la misma. Por ejemplo, este código genera una excepción `NullPointerException`:

```
Point blanco;  
System.out.println (blanco.x);
```

ArrayIndexOutOfBoundsException: El índice que se utiliza para acceder a un arreglo es negativo o mayor que `arreglo.length-1`. Si podés encontrar el sitio donde está el problema, agregá una sentencia de impresión inmediatamente antes, para imprimir el valor del índice y la longitud del arreglo. ¿El tamaño del arreglo es el adecuado? ¿El índice es correcto?

Ahora revisá tu programa y de dónde salen el arreglo y el índice. Buscá la sentencia de asignación más cercana y fijate si está haciendo lo correcto. Si cualquiera de ellos es un parámetro, andá al lugar donde se llama al método y fijate de dónde están viniendo los valores.

StackOverflowException: Exepción de desbordamiento de pila, ver “Recursividad infinita”.

He añadido tantas sentencias de impresión, que generaron un montón de salida

Uno de los problemas con el uso de sentencias de impresión para la depuración es que pueden terminar enterradas en la salida. Hay dos maneras de proceder: o simplificar la salida o simplificar el programa.

Para simplificar la salida, podés eliminar o comentar las sentencias de impresión que no ayudan, o combinarlas, o formatear la salida para que sea más fácil de entender. A medida que desarrollás un programa, vas a concebir formas de visualizar la ejecución del programa, y desarrollar código que genere de forma concisa la visualización de la información.

Para simplificar el programa, hay varias cosas que podés hacer. En primer lugar, reducir el problema del programa que está corriendo. Por ejemplo, si estás ordenando un arreglo, ordená un arreglo *pequeño*. Si el programa recibe entrada por el usuario, darle la entrada más simple que provoque el error.

En segundo lugar, limpiá el programa. Quitá código muerto y reorganizá el programa para que sea lo más fácil de leer, como sea posible. Por ejemplo, si sospechás que el error está en una parte profundamente anidada del programa, intentá volver a escribir esta parte con una estructura más sencilla. Si sospechás de un método grande, tratá de dividirlo en métodos más pequeños y ponerlos a prueba por separado.

A menudo el proceso de encontrar el mínimo caso de prueba te conduce al error. Por ejemplo, si encontrás que un programa funciona cuando el arreglo tiene un número par de elementos, pero no cuando se tiene un número impar, eso te da una pista sobre lo que puede estar pasando. Del mismo modo, la reescritura de un fragmento del código puede ayudar a encontrar errores sutiles. Si hacés un cambio que pensás que no afecta al programa, y lo hace, te puede dar una pista.

B.3 Errores semánticos

Mi programa no funciona

De alguna manera los errores semánticos son los más difíciles, porque el compilador y el sistema en tiempo de ejecución no proporcionan ninguna información sobre lo que está mal. Sólo vos sabés lo que el programa tenía que hacer, y sólo vos sabés que no lo está haciendo.

El primer paso es hacer una conexión entre el texto del programa y el comportamiento que estás viendo. Necesitás una hipótesis sobre lo que el programa está haciendo en realidad. Una de las cosas que hace que sea difícil es que las computadoras son muy rápidas. A menudo desearás que se pueda desacelerar el programa hasta la velocidad humana, pero no hay una forma directa de hacerlo, e incluso si la hubiera, no es realmente

un buena manera de depurar. Aquí hay algunas preguntas que deberías hacerte:

- ¿Hay algo que se suponía que el programa debe hacer, pero no parece estar pasando? Busca la sección del código que realiza esa función y asegurate de que se está ejecutando cuando pensás que debería hacerlo. Añadir una sentencia de impresión al principio de los métodos sospechosos.
- ¿Está ocurriendo algo que no debería? Busca el código en tu programa que realiza esa función y fijate si se está ejecutando cuando no debe.
- ¿Una sección de código produce un efecto que no es lo que esperabas? Asegurate de entender el código en cuestión, especialmente si se trata de llamadas a métodos preincorporados de Java. Lee la documentación para los métodos que se llaman. Probá los métodos llamándolos directamente con casos de prueba simples, y comprobá los resultados.

Finalizando el programa, necesitás tener un modelo mental de cómo funciona el programa. Si tu programa no hace lo que esperás, muchas veces el problema no está en el programa, está en tu modelo mental.

La mejor manera de corregir tu modelo mental es la de partir el programa en sus componentes (por lo general las clases y métodos) y probar cada componente de forma independiente. Una vez que encuentres la discrepancia entre tu modelo y la realidad, podés resolver el problema.

Por supuesto, debés construir y probar los componentes a medida que desarrolles el programa. Si surge algún problema, debe haber solamente una pequeña cantidad de código nuevo que no se sabe si es correcto. Estos son algunos errores semánticos comunes, que es posible que desees revisar:

- Si usás el operador de asignación, `=`, en lugar del operador de igualdad, `==`, en la sentencia de condición de un `if`, `while` o `for`, podrías obtener una expresión que es sintácticamente válida, pero no hace lo que esperás.
- Cuando se aplica el operador de igualdad, `==`, a un objeto, comprueba una igualdad superficial. Si deseás comprobar la igualdad de contenido, deberías usar el método `equals` (o definir uno, para los objetos definidos por el usuario).

- Algunas bibliotecas Java esperan que objetos definidos por el usuario definan métodos como `equals`. Si no los definís, heredarás el comportamiento predeterminado de la clase padre, que podrá no ser lo que estás necesitando.
- La herencia puede dar lugar a sutiles errores semánticos, porque se puede ejecutar código heredado sin que te des cuenta. Para asegurarte de que entendés el flujo de ejecución en su programa, consultá la sección titulada “Flujo de Ejecución”.

Tengo una expresión larga y muy fea y no hace lo que espero

Escribir expresiones complejas está bien siempre y cuando sean legibles, pero pueden ser difíciles de depurar. A veces es una buena idea partir una expresión compleja en una serie de asignaciones en variables temporales. Por ejemplo:

```
rect.setLocation (rect.getLocation().translate
                  (-rect.getWidth(), -rect.getHeight()));
```

puede ser reescrita como

```
int dx = -rect.getWidth();
int dy = -rect.getHeight();
Point ubicacion = rect.getLocation();
Point nuevaUbicacion = ubicacion.translate (dx, dy);
rect.setLocation (nuevaUbicacion);
```

La versión explícita es más fácil de leer, porque los nombres de las variables proporcionan documentación adicional, y es más fácil de depurar, ya que podemos comprobar los tipos de las variables intermedias y mostrar sus valores.

Otro problema que puede ocurrir con expresiones tan grandes es que el orden de evaluación puede no ser como esperás. Por ejemplo, si estás traduciendo la expresión $\frac{x}{2\pi}$ en Java, es posible escribir

```
double y = x / 2 * Math.PI;
```

Esto no es correcto, porque la multiplicación y la división tienen la misma precedencia y se evalúan de izquierda a derecha. Así que esta expresión calcula $x\pi/2$. Una buena forma de depurar expresiones es agregar paréntesis para hacer el orden de evaluación explícita.

```
double y = x / (2 * Math.PI);
```

Cada vez que no estés seguro del orden de evaluación, usá los paréntesis. No sólo el programa será correcto (en el sentido de hacer lo que querés), sino que también será más comprensible para otras personas que no hayan aprendido de memoria las reglas de precedencia.

Tengo un método que no devuelve lo que espero

Si tenés una sentencia de retorno con una expresión compleja, no tenés la oportunidad de imprimir el valor de retorno antes de que lo devuelva. Una vez más, podés utilizar una variable temporal. Por ejemplo, en lugar de

```
public Rectangle interseccion (Rectangle a, Rectangle b) {  
    return new Rectangle (  
        Math.min (a.x, b.x),  
        Math.min (a.y, b.y),  
        Math.max (a.x+a.width, b.x+b.width)-Math.min (a.x, b.x)  
        Math.max (a.y+a.height, b.y+b.height)-Math.min (a.y, b.y) );  
}
```

se puede escribir

```
public Rectangle interseccion (Rectangle a, Rectangle b) {  
    int x1 = Math.min (a.x, b.x);  
    int y2 = Math.min (a.y, b.y);  
    int x2 = Math.max (a.x+a.width, b.x+b.width);  
    int y2 = Math.max (a.y+a.height, b.y+b.height);  
    Rectangle rect = new Rectangle (x1, y1, x2-x1, y2-y1);  
    return rect;  
}
```

Ahora tenés la oportunidad de mostrar cualquiera de las variables intermedias antes de devolver el resultado.

Mi sentencia de impresión no está haciendo nada

Si utilizás el método `println`, la salida se muestra inmediatamente, pero si utilizás `print` (al menos en algunos ambientes), la salida se almacena sin que se muestre hasta que salga el carácter de nueva línea. Si el programa termina sin producir una nueva línea, puede que nunca vea la salida almacenada. Si sospechás que esto te está sucediendo, tratá de cambiar parte o la totalidad de la sentencia `print` a `println`.

Estoy muy, muy atascado y necesito ayuda

En primer lugar, tratá de alejarte de la computadora por unos minutos. Las computadoras emiten ondas que afectan al cerebro, causando los siguientes síntomas:

- frustración y/o rabia.
- creencias supersticiosas (“la computadora me odia”) y pensamiento mágico (“el programa sólo funciona cuando me pongo mi sombrero al revés”).

- programación sobre la marcha al azar (intentar programar escribiendo todo programa que sea posible y elegir uno que haga las cosas bien).

Si te encontrás sufriendo cualquiera de estos síntomas, levántate y andá a dar un paseo. Cuando estés calmado, pensá sobre el programa. ¿Qué está haciendo? ¿Cuáles son las posibles causas de su comportamiento? ¿Cuándo fue la última vez que tenías un programa funcionando, y ¿qué hiciste después?

A veces sólo se necesita tiempo para encontrar un error. A menudo encuentro errores cuando estoy lejos de la computadora y dejo a mi mente vagar. Algunos de los mejores lugares para encontrar los errores son los trenes, las duchas, y en la cama, justo antes de dormirse.

No, realmente necesito ayuda

Sucede. Incluso los mejores programadores de vez en cuando se atascan. A veces se trabaja en un programa durante mucho tiempo, que no podés ver el error. Un par de ojos frescos es justo lo que hace falta.

Antes de traer a otra persona, asegurate de haber agotado las técnicas descritas aquí. Tu programa debe ser tan simple como sea posible, y deberías estar trabajando con entradas tan pequeñas como sean posibles que causen el error. Debés tener sentencias de impresión en los lugares apropiados (y la salida que producen debe ser comprensible). Debés entender el problema lo suficientemente bien para describirlo en forma concisa.

Al traer a alguien que te ayude, asegurate de darle la información que necesita.

- ¿Qué tipo de error es este? ¿En tiempo de compilación, en tiempo de ejecución, o semántico?
- Si el error se produce en tiempo de compilación o en tiempo de ejecución, ¿cuál es el mensaje de error, y qué parte del programa se está indicando?
- ¿Qué fue lo último que hizo antes de que ocurra este error? ¿Cuáles fueron las últimas líneas de código que escribiste, o cuál es el nuevo caso de prueba que falla?
- ¿Qué intentaste hasta ahora?, y ¿qué aprendiste?

A menudo, encontrarás que mientras le estás explicando el problema a otra persona, encontrarás la respuesta. Este fenómeno es tan común que algunas personas recomiendan una técnica de depuración llamada “el pato de goma.” Así es cómo funciona:

1. Comprar un pato de goma común.
2. Cuando estás realmente atascado en un problema, poné el pato de goma en la mesa en frente tuyo y decile: “pato de goma, me estoy atascado en un problema. Esto es lo que está pasando ...”
3. Explicá el problema al pato de goma.
4. Vé la solución.
5. Agradecer al pato de goma.

¡Encontré el error!

Muy a menudo, cuando encontrás un error, es evidente cómo solucionarlo. Pero no siempre. A veces lo que parece ser un error es en realidad una indicación de que no entendés muy bien el programa, o hay un error en tu algoritmo. En estos casos, podría tener que replantearse el algoritmo, o ajustar el modelo mental del programa. Tomate un tiempo fuera de la computadora para pensar en el programa, trabajá a través de algunos casos de prueba a mano o dibujá diagramas para representar el cálculo.

Al corregir un error, no te sumerjas y empieces a cometer nuevos errores. Tomá un segundo para pensar en qué clase de error era, ¿por qué cometiste el error en primer lugar?, ¿cómo se manifestó el error?, y ¿qué podrías haber hecho para encontrarlo más rápido? La próxima vez que veas algo similar, podrás encontrar el error más rápidamente.

Apéndice C

Entrada y salida en Java

C.1 Objetos del sistema

`System`¹ es el nombre de la clase preincorporada que contiene métodos y objetos usados para obtener datos desde el teclado, imprimir texto a través de la pantalla y manejar la entrada/salida (E/S) por archivos.

`System.out` es el nombre del objeto que usamos para mostrar texto en la pantalla. Cuando invocás a `print` y `println`, lo hacés con el objeto llamado `System.out`. Es interesante ver que se puede imprimir `System.out`:

```
System.out.println (System.out);
```

La salida será:

```
java.io.PrintStream@80cc0e5
```

Como de costumbre, cuando Java imprime un objeto, se imprime el tipo del objeto, el cual es `PrintStream`, el paquete en el cual ese tipo está definido, `java.io`, y un identificador único para el objeto. En mi máquina el identificador es `80cc0e5`, pero si vos ejecutás el mismo código, probablemente vas a obtener algo diferente.

También existe un objeto llamado `System.in`; este objeto es de tipo `BufferedInputStream`. El objeto `System.in` hace posible obtener datos a través del teclado. Desafortunadamente, no es fácil obtener datos a través del teclado.

1. N.d.T: *Sistema* en inglés.

C.2 Entrada por teclado

Primero que nada, tenemos que usar `System.in` para crear un nuevo objeto de tipo `InputStreamReader`.

```
InputStreamReader entrada = new InputStreamReader (System.in);
```

Luego usamos `entrada` para crear un nuevo objeto `BufferedReader`:

```
BufferedReader teclado = new BufferedReader (entrada);
```

El objetivo de toda esta manipulación es que existe un método que podemos llamar en un `BufferedReader`, llamado `readLine`², que toma datos del teclado y los convierte en una cadena. Por ejemplo:

```
String s = teclado.readLine ();  
System.out.println (s);
```

lee una línea a través del teclado e imprime el resultado.

Existe sólo un problema; hay cosas que pueden salir mal cuando se llama a `readLine`, y pueden causar una excepción de tipo `IOException`. Hay una regla en Java que indica que si un método puede causar una excepción, debe aclararlo. La sintaxis se ve así:

```
public static void main (String[] args) throws IOException {  
    // cuerpo del main  
}
```

Esto indica que el método `main` puede “lanzar”³ una excepción de tipo `IOException`. Podés pensar el hecho de lanzar una excepción como si estuviese armando un berrinche.

C.3 Entrada por archivos

Leer datos de entrada desde un archivo es igualmente estúpido. Acá hay un ejemplo:

```
public static void main (String[] args) throws  
    FileNotFoundException, IOException {  
    procesarArchivo ("/usr/dict/palabras.txt");  
}
```

2. N.d.T.: *leer línea* en inglés.

3. N.d.T.: Esta es la traducción del verbo *throw* que aparece en la declaración del `main`.

```

public static void procesarArchivo (String nombreadarchivo)
    throws FileNotFoundException, IOException {

    FileReader lector = new FileReader (nombreadarchivo);
    BufferedReader entrada = new BufferedReader (lector);

    while (true) {
        String s = entrada.readLine();
        if (s == null) break;
        System.out.println (s);
    }
}

```

Este programa lee cada línea del archivo que se encuentra en la ubicación `/use/dict/palabras.txt` y las almacena en una cadena para luego imprimirlas por pantalla. Nuevamente, la declaración en el encabezado `throws FileNotFoundException, IOException` es requerida por el compilador. Los tipos de objeto `FileReader` y `BufferedReader` son parte de la increíblemente complicada jerarquía de clases que Java usa para hacer cosas increíblemente comunes y simples. Más allá de eso, no hay mucho más que decir sobre cómo funciona este fragmento de código.

Apéndice D

Gráficos

D.1 Objetos Pizarra y Graphics

Hay varias maneras de crear gráficos en Java, algunas más complicadas que otras. Para hacerlo fácil, he creado un objeto llamado `Pizarra` que representa una superficie en la que podés dibujar. Cuando se crea una `Pizarra`, aparece una nueva ventana en blanco. La `Pizarra` contiene un objeto `Graphics`, que se usa para dibujar en la pizarra.

Los métodos pertenecientes a los objetos de `Graphics` están definidos en la clase preincorporada `Graphics`. Los métodos que pertenecen a `Pizarra` están definidos en la clase `Pizarra`, que se muestra en la Sección D.6. Usá el operador `new` para crear nuevos objetos `Pizarra`:

```
Pizarra pizarra = new Pizarra (500, 500);
```

Los parámetros son el ancho y el alto de la ventana. El valor de retorno es asignado a una variable llamada `pizarra`. No existe conflicto entre el nombre de la clase (con “P” mayúscula) y el nombre de la variable (con “p” minúscula).

El siguiente método que necesitamos es `dameGraphicsDePizarra`, que devuelve un objeto `Graphics`. Podés pensar a un objeto `Graphics` como un trozo de tiza.

```
Graphics g = pizarra.dameGraphicsDePizarra();
```

Usamos el nombre `g` por convención, pero podríamos haberlo llamado de cualquier manera.

D.2 Llamando métodos en un objeto Graphics

A fin de dibujar cosas en la pantalla, se llaman métodos del objeto `graphics`.

```
g.setColor (Color.black);
```

`setColor`¹ cambia el color actual, en este caso a negro. Todo lo que se dibuje será negro, hasta que se use `setColor` de nuevo.

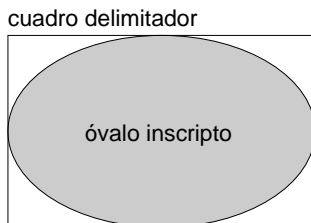
`Color.black` es un valor especial provisto por la clase `Color`, del mismo modo en que `Math.PI` es un valor especial provisto por la clase `Math`. Te debe hacer feliz saber que `Color` provee toda una paleta de colores, incluyendo:

<code>black</code>	<code>blue</code>	<code>cyan</code>	<code>darkGray</code>	<code>gray</code>	<code>lightGray</code>
<code>magenta</code>	<code>orange</code>	<code>pink</code>	<code>red</code>	<code>white</code>	<code>yellow</code>

Para dibujar en la Pizarra, podemos llamar a los métodos `draw`² sobre el objeto `Graphics`. Por ejemplo:

```
g.drawOval (x, y, ancho, alto);
```

`drawOval`³ toma cuatro enteros como parámetros. Estos parámetros especifican un **cuadro delimitador**, que es el rectángulo dentro del cual estará dibujado el óvalo (como se muestra en la figura). El cuadro delimitador en sí mismo no es dibujado; solamente el óvalo. El cuadro delimitador es sólo una guía. Los cuadros delimitadores siempre se orientan horizontal o verticalmente; nunca están dispuestos en diagonal.



Si lo pensás, existen muchas maneras de especificar la ubicación y el tamaño de un rectángulo. Podrías darle la ubicación del centro o cualquiera de las esquinas, junto con el alto y el ancho. O podrías, en cambio,

1. N.d.T.: `assignColor`.

2. N.d.T.: `dibujar`.

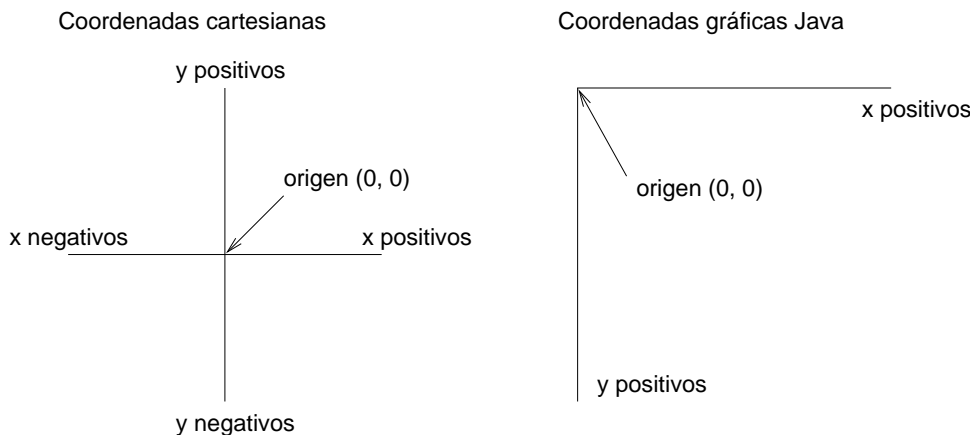
3. N.d.T.: `dibujarOvalo`.

darle la ubicación de las esquinas opuestas. La elección es arbitraria, pero en cualquier caso requerirá la misma cantidad de parámetros: cuatro.

Por convención, la manera usual de especificar un cuadro delimitador, es dar la ubicación de la esquina *superior-izquierda* y el alto y ancho. La manera usual de especificar la ubicación es usando un **sistema de coordenadas**.

D.3 Coordenadas

Probablemente estés familiarizado con las coordenadas cartesianas en dos dimensiones, en las que cada posición se identifica por una coordenada x (distancia en el eje x) y una coordenada y . Por convención, las coordenadas cartesianas crecen hacia la derecha y hacia arriba, como se ve en la figura.



Para fastidio de todos, en los sistemas de computación gráfica es convención usar una variante de las coordenadas cartesianas en las que el origen está en el rincón superior-izquierdo de la pantalla o de la ventana, y el eje y positivo crece hacia *abajo*. Java sigue esta convención.

La unidad de medida se llama **pixel**; una pantalla típica mide unos 1000 píxeles de ancho. Las coordenadas son siempre enteros. Si querés usar valores de punto flotante como coordenadas, deberás redondearlos a enteros (Ver Sección 3.2).

D.4 Un Ratón Mickey medio pelo

Supongamos que queremos hacer un dibujo del Ratón Mickey. Podemos usar el óvalo que acabamos de dibujar como la cara, y luego agregarle las orejas. Antes de hacer eso es una buena idea partir el programa en dos métodos. `main` creará el objeto `Pizarra` y el objeto `Graphics` y luego llamará a `dibujar`, quien efectivamente dibuja.

Cuando terminamos de llamar a los métodos `draw`, debemos llamar `pizarra.repaintar` para hacer aparecer los cambios en pantalla.

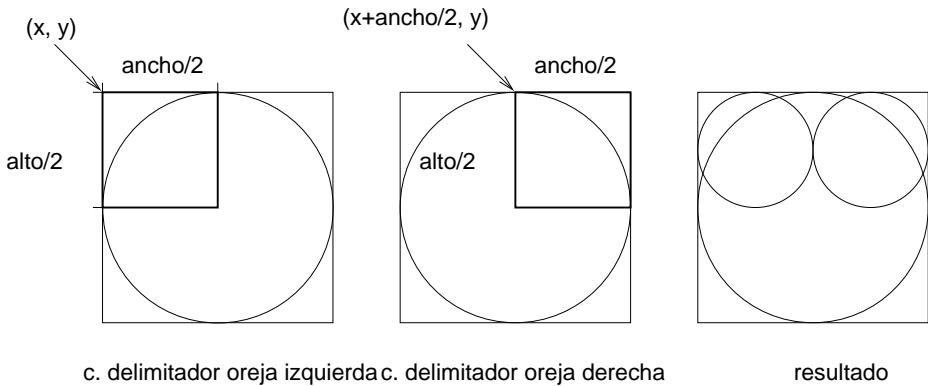
```
public static void main (String[] args) {
    int ancho = 500;
    int alto = 500;

    Pizarra pizarra = Pizarra.crearPizarra (ancho, alto);
    Graphics g = Pizarra.dameGraphics (pizarra);

    g.setColor (Color.black);
    dibujar (g, 0, 0, ancho, alto);
    pizarra.repaintar ();
}

public static void dibujar
    (Graphics g, int x, int y, int ancho, int alto) {
    g.drawOval (x, y, ancho, alto);
    g.drawOval (x, y, ancho/2, alto/2);
    g.drawOval (x+ancho/2, y, ancho/2, alto/2);
}
```

Los parámetros para dibujar son el objeto `Graphics` y un cuadro delimitador. `dibujar` llama a `drawOval` tres veces, para dibujar la cabeza de Mickey y dos orejas. La siguiente figura muestra los cuadros delimitadores de las orejas.



Como se ve en la figura, las coordenadas de la esquina superior - izquierda del cuadro delimitador de la oreja izquierda son (x, y) . Las coordenadas para la oreja derecha son $(x+ancho/2, y)$. En ambos casos, el ancho y alto de las orejas es la mitad del ancho y alto del cuadro delimitador original.

Notá que las coordenadas de los cuadros de las orejas son todos relativos a la posición $(x$ e $y)$ y el tamaño (ancho y alto) del cuadro delimitador original. Como resultado, podemos usar dibujar para dibujar al Ratón Mickey (aunque sea uno medio pelo) en cualquier lugar de la pantalla y de cualquier tamaño.

Ejercicio D.1

Modificá los parámetros pasados a dibujar de forma tal que Mickey sea la mitad del alto y ancho de la pantalla, y centrado.

D.5 Otros comandos para dibujar

Otro comando para dibujar con los mismos parámetros que `drawOval` es

```
drawRect (int x, int y, int ancho, int alto)
```

Aquí estoy usando un formato estándar para documentar el nombre y los parámetros de los métodos. Esta información es en ocasiones la **interfaz** del método o el **prototipo**. Mirando este prototipo, podés saber de qué tipo son los parámetros y (basado en sus nombres) inferir qué hacen. Aquí hay otro ejemplo:

```
drawLine (int x1, int y1, int x2, int y2)
```

El uso de `x1`, `x2`, `y1` e `y2` como nombres de parámetros sugiere que el método `drawLine` dibuja una línea desde el punto (`x1`, `y1`) hasta el punto (`x2`, `y2`). Otro comando que podrías querer probar es

```
drawRoundRect (int x, int y, int ancho, int alto,  
               int arcAncho, int arcAlto)
```

Los primeros cuatro parámetros especifican el cuadro delimitador del rectángulo; los dos parámetros restantes indican qué tan redondeadas deberán ser las esquinas, especificando el diámetro de los arcos en las esquinas.

Existen además versiones “fill”⁴ de estos comandos, que no sólo dibujan el borde de una figura, sino también la rellenan. Las interfaces son idénticas; sólo cambian los nombres:

```
fillOval (int x, int y, int ancho, int alto)  
fillRect (int x, int y, int ancho, int alto)  
fillRoundRect (int x, int y, int ancho, int alto,  
               int arcAncho, int arcAlto)
```

No existe nada del estilo `fillLine`—simplemente no tiene sentido.

D.6 La clase Pizarra

```
import java.awt.*;  
  
class Ejemplo {  
  
    // demuestra un uso simple de la clase Pizarra  
  
    public static void main (String[] args) {  
        int ancho = 500;  
        int alto = 500;  
  
        Pizarra pizarra = new Pizarra (ancho, alto);  
        Graphics g = pizarra.dameGraphicsDePizarra ();  
  
        g.setColor (Color.blue);  
  
        dibujar (g, 0, 0, ancho, alto);  
        pizarra.repaint ();  
  
        anim (pizarra);  
    }  
}
```

4. N.d.T.: “rellenas”.

```

// dibujar demuestra un patrón recursivo

public static void dibujar (Graphics g, int x, int y,
                           int ancho, int alto) {
    if (alto < 3) return;

    g.drawOval (x, y, ancho, alto);

    dibujar (g, x, y+alto/2, ancho/2, alto/2);
    dibujar (g, x+ancho/2, y+alto/2, ancho/2, alto/2);
}

// anim demuestra una animación simple

public static void anim (Pizarra pizarra) {
    Graphics g = pizarra.image.getGraphics ();
    g.setColor (Color.red);

    for (int i=-100; i<500; i+=10) {
        g.drawOval (i, 100, 100, 100);
        pizarra.repaint ();
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {}
    }
}

class Pizarra extends Frame {

    // imagen es un buffer: cuando los usuarios de Pizarra
    // dibujen cosas, lo harán en el buffer. Cuando la
    // Pizarra se pinte, copiaremos la imagen en la pantalla.
    Image imagen;

    public Pizarra (int ancho, int alto) {
        setBounds (100, 100, ancho, alto);
        setBackground (Color.white);
        setVisible (true);
        imagen = createImage (ancho, alto);
    }

    // cuando un usuario de la Pizarra pide el objeto Graphics,
    // le devolvemos el del buffer, no el de la pantalla

```



```

public Graphics getSlateGraphics () {
    return imagen.getGraphics ();
}

// normalmente el método update borra la pantalla
// y llama a paint, pero dado que estamos sobrescribiendo
// la pantalla de un modo u otro, es ligeramente más rápido
// sobrescribir update y evitarse limpiar la pantalla

public void update (Graphics g) {
    paint (g);
}

// paint copia el buffer a la pantalla

public void paint (Graphics g) {
    if (imagen == null) return;
    g.drawImage (imagen, 0, 0, null);
}
}

```

Ejercicio D.2

El propósito de este ejercicio es practicar el uso de métodos como una forma de organizar y encapsular tareas complejas.

ADVERTENCIA: Es de suma importancia que realices este ejercicio de a un paso por vez, y que tengas cada paso funcionando correctamente antes de continuar. Más importante aun, asegúrate de entender cada paso antes de continuar.

- a. Creá un nuevo programa llamado `PersonaDeNieve.java`. Copiá el código del archivo <https://sourceforge.net/projects/thinkcsjava2esp/files/PersonaDeNieve.java>. Correló. Debería aparecer una nueva ventana con un puñado de círculos azules y otro puñado de círculos rojos. Esta ventana es la Pizarra.

A diferencia de los otros programas que corrimos, en este no hay ventana de consola. De todos modos, si agregás una sentencia `print` o `println`, la consola aparecerá. En las aplicaciones gráficas, la consola es útil para depurar.

Noté que no puedo cerrar la ventana Pizarra haciendo click en ella, lo cual es probablemente bueno, porque te recordará salir desde el intérprete cada vez que corras el programa.

- b. Echale un vistazo al código como está actualmente y asegúrate de entender todo el código de dibujar. El método llamado `anim` está ahí para tu entretenimiento, pero no lo estaremos usando para este ejercicio. Deberías quitarlo.

- c. La clase Pizarra aparece inmediatamente después de la clase Ejemplo. Tal vez quieras revisarla, aunque mucho de lo que hay ahí no tendrá sentido a esta altura.
- d. Probá cambiar las sentencias de dibujar y ver qué efectos tienen esos cambios. Probá los diferentes comandos de dibujado. Para más información acerca de ellos, mirá <http://java.sun.com/products/jdk/1.1/docs/api/java.awt.Graphics.html>
- e. Cambiá el alto o el ancho de la Pizarra y corré el programa nuevamente. Deberías ver que la imagen ajusta su tamaño y su proporción para caber perfectamente dentro de la ventana. Escribirás programas dibujadores que hagan esa misma cosa. La idea es usar variables y parámetros para hacer programas más generales; en este caso la generalidad es que podamos dibujar imágenes de cualquier tamaño y en cualquier ubicación.

Usá un Cuadro Delimitador

Los parámetros que recibe dibujar (sin incluir el objeto graphic) conforman un **cuadro delimitador**. El cuadro delimitador especifica el rectángulo invisible en el que dibujar debería dibujar.

- a. Dentro de dibujar, creá un nuevo cuadro delimitador que tenga la misma altura que la Pizarra pero sólo un tercio de su ancho, y esté centrada. Cuando digo “creá un cuadro delimitador” me refiero a que definas cuatro variables que contengan la ubicación y el tamaño. Deberás pasar este cuadro delimitador como parámetro a dibujarPersonaDeNieve.
- b. Creá un nuevo método llamado dibujarPersonaDeNieve que tome los mismos parámetros que dibujar. Para empezar, deberá dibujar un simple óvalo que rellene completamente al cuadro delimitador (en otras palabras, deberá tener la misma posición y ubicación que el cuadro delimitador).
- c. Cambiá el tamaño de la Pizarra y corré el programa de nuevo. El tamaño y la proporción del óvalo deberían ajustarse de modo que, sin importar cuál sea el tamaño de la Pizarra, el óvalo tenga la misma altura, un tercio del ancho, y esté centrado.

Hacer una Persona de Nieve

- a. Modificá dibujarPersonaDeNieve para que dibuje tres óvalos apilados uno sobre el otro como un muñeco de nieve. La altura y el ancho de la persona de nieve deberían llenar el cuadro delimitador.
- b. Cambiá el tamaño de la Pizarra y corré el programa nuevamente. De nuevo, la persona de nieve debería ajustarse de forma que siempre toque el tope y la base de la pantalla, y los tres óvalos se toquen, y que la proporción de los tres óvalos se mantenga igual.
- c. A esta altura, mostrale el programa a tu profesor para asegurarte de haber entendido las ideas básicas detrás de este ejercicio. No deberías continuar trabajando en esto hasta que no hayas hecho revisar tu programa.

Dibujar la Persona de Nieve Gótico Estadounidense

- a. Modificá dibujar de modo tal que dibuje dos personas de nieve uno al lado del otro. Uno de ellos deberá ser tan alto como la ventana; el otro deberá tener 90 % del alto de la ventana. (La gente de nieve muestra dimorfismo sexual en el que las hembras son aproximadamente 10 % más pequeñas que los machos.)

Sus anchos deberán ser proporcionales a sus altos, y sus cuadros delimitadores deberán ser adyacentes (lo que significa que los óvalos dibujados probablemente no se toquen). La pareja deberá estar centrada, queriendo decir que deberá haber el mismo espacio a cada lado.

Con una pipa de marlo...

- a. Escribí un método llamado dibujarCara que tome el mismo número de parámetros que dibujarPersonaDeNieve, y que dibuje una simple cara dentro del cuadro delimitador dado. Dos ojos serán suficientes, pero podés hacerlo tan elaborado como gustes.

De nuevo, a medida que redimensiones la ventana, las proporciones de la cara no deberían cambiar, y todas las características deberían caber dentro del óvalo (exceptuando posiblemente a las orejas).

- b. Modificá el dibujarPersonaDeNieve de modo que llame a dibujarCara. El cuadro delimitador que le pases a dibujarCara deberá ser el mismo que aquel que creó el óvalo de la cara.

Ponéle camisetas a las personas de nieve

- a. Agregá un nuevo método llamado dibujarCamiseta que tome los parámetros usuales y que dibuje algún tipo de logo de camiseta dentro del cuadro delimitador. Podés usar cualquiera de los comandos de dibujo, aunque tal vez prefieras evitar el uso de drawString (dibujar cadena) dado que no es fácil garantizar que la cadena cabrá dentro del cuadro delimitador.
- b. Modificá dibujarPersonaDeNieve para que llame a dibujarCamiseta a fin de estamparle algo en el pecho (el óvalo del central) de cada PersonaDeNieve.

Hacé que la Señora de Nieve esté embarazada

- a. Me di cuenta de que esto es un poco riesgoso, pero tiene un objetivo. Modificá dibujar de modo que luego de dibujar a la Sra. De Nieve, dibuje un pequeño Bebé de Nieve dentro de su óvalo abdominal.

Notá que agregar algo a dibujar afecta solamente a una persona de nieve. Agregar algo a dibujarPersonaDeNieve afecta a todas las personas de nieve.

Hacé que todas las personas de nieve estén embarazadas

Bueno, ahora imaginemos que en vez de hacer que una sola persona de nieve esté embarazada, queremos que todas lo estén. En lugar de agregar una línea a dibujar, agregaríamos una línea a dibujarPersonaDeNieve. ¿Qué haría esa línea? ¿Llamaría a dibujarPersonaDeNieve! ¿Es posible hacer eso? ¿Es posible llamar a

un método desde adentro de sí mismo? Bueno, sí, se puede, pero debés hacerlo con cuidado.

¡¡¡ASÍ QUE AÚN NO LO HAGAS!!!

Pensá por un minuto. Qué pasaría si dibujamos una persona de nieve grande, y luego pusiéramos una persona de nieve chica en su óvalo abdominal. Luego tendríamos que poner una persona de nieve aun más chica en el óvalo abdominal pequeño, y así siguiendo. El problema es que nunca pararía. Seguiríamos dibujando personas de nieve chicas y más chicas hasta el día del juicio final.

Una solución es elegir un tamaño mínimo de persona de nieve, y decir que por debajo de ese tamaño, nos rehusamos a dibujar más personas de nieve.

- a. Agregá una línea al comienzo del método `dibujarPersonaDeNieve` que se fije la altura del cuadro delimitador y retorne inmediatamente si es menos que 10.

```
if (altura < 10) return;
```

- b. Ahora que tenés esa línea, es seguro agregar código al final del método de modo que luego de dibujar los óvalos, y la cara, y la camiseta, llame a `dibujarPersonaDeNieve` para poner un hombrecito de nieve en el abdomen.

Ejercicio D.3

- a. Creá un nuevo programa llamado `Moire.java`.
- b. Agregá el siguiente código a tu proyecto, y reemplazá los contenidos de dibujar con una simple línea que llame a `moire`.

```
public static void moire(Graphics g, int x, int y,
                        int ancho, int alto) {
    int i = 1;
    while (i<ancho) {
        g.drawOval (0, 0, i, i);
        i = i + 2;
    }
}
```

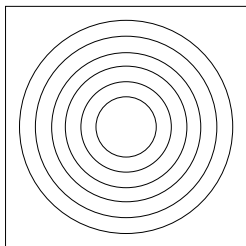
- c. Mirá el código antes de correrlo y dibujá un borrador de lo que esperarías que haga. Ahora corrélo. ¿Obtuviste lo que esperabas? Para tener una explicación parcial de lo que está ocurriendo, mirá lo que sigue:

http://es.wikipedia.org/wiki/Patr%C3%B3n_de_muar%C3%A9
<http://vicente1064.blogspot.com/2009/12/espectaculares-patrones-de-moire-que-se.html>

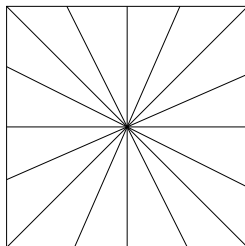
- d. Modificá tu programa para que el espacio entre los círculos sea más grande o más chico. Fijate qué le sucede a la imagen.

- e. Modificá el programa para que los círculos se dibujen en el centro de la pantalla y de forma concéntrica, como se ve en la siguiente figura. A diferencia de la figura, la distancia entre los círculos debería ser lo suficientemente chica como para que la interferencia del Muaré sea visible.

círculos concéntricos



patrón radial



- f. Escribí un método llamado `radial` que dibuje un conjunto radial de segmentos de líneas como se ve en la figura, pero deberán estar lo suficientemente cerca unos de otros como para crear un patrón Muaré.
- g. Casi cualquier tipo de patrón gráfico puede generar patrones de interferencia tipo Muaré. Jugá y fijate lo que podés crear.