Introducción a las Ciencias de la Computación

Manual de Prácticas 2018-I

(Sin terminar)

Verónica E. Arriola-Ríos

Facultad de Ciencias,
UNAM

Índice general

Parte I
Prácticas

1 | Compiladores y intérpretes

Definición 1.1: Compilador

"Un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje *fuente*, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje *objeto*. Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de [algunos] errores en el programa fuente."[a]

[a] Aho, Sethi y Ullman 1998, pp. 1.

Definición 1.2: Intérprete

Un intérprete es un programa que, una vez cargado en la memoria de una computadora y al ejecutarse, procede como sigue:

1. Toma un enunciado del programa en lenguaje de alto nivel, llamado código fuente.
2. Traduce ese enunciado y lo ejecuta.
3. Repite estas dos acciones hasta que alguna instrucción le indique que pare, o bien tenga un error fatal en la ejecución.

Meta

Que el alumno aprenda a utilizar un compilador para traducir código, detectar y corregir errores sintácticos y semánticos; y un intérprete para ejecutar `bytecode`.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Invocar al compilador de `Java`, `javac` desde una terminal para generar el *byte-code* ejecutable por la máquina virtual.
- Invocar a la máquina virtual de `Java` con el comando `java`, para ejecutar al código de una clase que contenga un método `main`.
- Empaquetar los archivos resultantes utilizando el comando `jar`.
- Ejecutar código en el archivo `.jar`.
- Generar la documentación del paquete utilizando `javadoc`.
- Utilizar la herramienta `ant` para compilar y empaquetar código, generar documentación y ejecutar un programa, utilizando un archivo `build.xml` provisto.

Desarrollo

Los programas de la JDK

Actividad 1.1

Descarga el archivo `practica1.tar.gz` y descomprímelo con el comando:

```
$ tar zxvf practica1.tar.gz
```

En el interior encontrarás dos carpetas: `Entrada` y `Reloj`. La primera parte de esta práctica se realizará con los archivos que se encuentran dentro de `Reloj`.

Actividad 1.2

Entra al directorio `src` y ejecuta los comandos siguientes:

```
1 $ javac icc/practica1/UsoReloj.java
```

Verás que aparecen una serie de errores, todos en el archivo `UsoReloj.java`. Abre ese archivo y utiliza los mensajes de error que te dio el compilador para corregir los errores. Puedes deducir con lógica qué es lo que debes hacer para lograrlo. Tendrás que invocar al compilador tantas veces como sea necesario hasta que ya

no aparezcan errores.

Una vez que ya no haya errores, notarás que aparecieron archivos con terminación `.class` dentro de la carpeta `icc1/practica1`. Notarás que esos archivos tienen los mismos nombres que los archivos `.java` originales. Estos nuevos archivos contienen *bytecode*. Si intentas abrirlos con un editor de texto obtendrás una serie de símbolos ininteligibles, eso si no trabas a tu editor.

Escribe exactamente qué archivos fueron creados y dónde.

Actividad 1.3

Ahora invoca a la máquina virtual de `Java` para que interprete el código que generaste.

```
$ java icc.practica1.UsoReloj
```

Listo, ya tienes tu primer programa corriendo.

Actividad 1.4

Intenta invocar a la máquina virtual con los nombres de otros archivos `.class`. ¿Qué sucede? Lee lo que devuelve la consola y abre los archivos `.java` correspondientes que necesites. ¿Qué tiene el archivo `UsoReloj.java` que permite invocarl su `.class` con `java`?

Actividad 1.5

Ahora crearemos un archivo comprimido con solo el código ejecutable:

```
$ jar cvf UsoReloj.jar icc/practica1/*.class
```

Puedes ejecutar el programa almacendo en el `.jar` con:

```
$ java -cp UsoReloj.jar icc.practica1.UsoReloj
```

La opción `-cp` es una abreviatura de `classpath` y se usa para indicar a `Java` dónde buscar los archivos `.class`.

Para más información sobre el uso de `jar`, visita el `tutorial oficial`.

Actividad 1.6

Finalmente, ejecuta el comando siguiente:

```
$ javadoc icc.practica1
```

Esto generará una serie de archivos `.html` en el directorio donde te encuentras. Usa tu navegador de internet y abre el que se llama `index.html`. ¿Qué observas? Aquí están todos los comentarios en los archivos de código con los que estás trabajando, pero en un formato más amigable para el lector. Esta es la documentación que le darás a tus usuarios cuando entregues tus trabajos.

Usando una herramienta auxiliar: ant

Aunque generaste todo lo necesario, el código compilado y la documentación se mezcló con los archivos con el código fuente. Al entregar un trabajo esto se ve desordenado y también te hará a tí más complicado el organizar tus archivos. A continuación lee la redacción de esta misma práctica del manual del Canek Peláez y Elisa Viso, para que repitas las mismas tareas, pero utilizando la herramienta `ant`. Las correcciones que realizaste al archivo `UsoReloj.java` son las mismas, así que guárdalas porque tendrás que entregar ese archivo. Cuando termines recuerda responder las preguntas que ahí se indican.

Ojo, tu archivo `build.xml` ha sido modificado ligeramente, por lo que los comandos con los que cuentas son:

compile	Compila la práctica.
run	Ejecuta la práctica, compilándola si no ha sido compilada.
doc	Genera la documentación JavaDoc de la práctica.
clean	Limpia la práctica de *bytecode*, documentación, o ambos.

Estructura de un programa

Para esta parte utilizarás el código dentro de la carpeta `Entrada`. El objetivo de esta sección es explicar cómo luce un programa sencillo en `Java` y descifrar algo del código que viste en los archivos de los ejercicios anteriores.

Juntos, el compilador `javac` y el intérprete `java` transforman el código que escribiste en secuencias de comandos que la computadora puede ejecutar como un programa.

Los programas de `Java` son muy parecidos a los programas que utilizas en la consola de Linux, como `ls`, `cd`, `pwd`, `more`, etc. En particular, también les puedes en-

viar parámetros al invocarlos, como harías al llamar `ls -al` o `diff archivo.txt archivo2.txt`.

Actividad 1.7

Entra a la carpeta `Entrada` e invoca `ant`. Te darás cuenta de que generó un archivo extra: `Entrada.jar`. Ignóralo por el momento y entra a la carpeta `build`. Usar los conocimientos adquiridos en las secciones anteriores para ejecutar desde aquí el programa `Entrada`. ¿Qué mensaje aparece?

Ahora ejecuta:

```
1 $ java icc.entrada.Entrada arg1 arg2 arg3
```

¿Qué obtienes?

Los archivos `.class` que se encuentran dentro de `build` fueron empaquetados en archivo `Entrada.jar`, por lo que es posible ejecutar el mismo programa utilizando sólo el `.jar`.

Actividad 1.8

Regresa al directorio que contiene el archivo `Entrada.jar` y ejecuta:

```
1 $ java -jar Entrada.jar arg1 arg2 arg3
```

Prueba con diferentes argumentos y reporta lo que hace el programa.

Actividad 1.9

Ahora abre el código en `src/icc/entrada/Entrada.java` y lee cuidadosamente su documentación. Modifica el texto que se produce con cada argumento recibido. Entregarás tu archivo modificado como parte de esta práctica.

Actividad 1.10

Lee los dos archivos `build.xml` utilizados en esta práctica y observa en qué se parecen y en qué difieren. ¿Qué objetivos reconoce cada archivo? ¿Qué pasos ejecutará cada uno de los objetivos (observa el atributo llamado *depends*)?

---

2 | Tipos primitivos y bits

En el momento en que se elige utilizar un sistema físico para realizar cómputos, éstos quedan sujetos a las leyes que rigen estos sistemas. Al elegir utilizar sistemas binarios con dos estados distinguibles, dichos cómputos quedan encuadrados por la lógica booleana; más aún, por una lógica booleana con un número grande, pero finito de símbolos disponibles.

Varios lenguajes de programación nos permiten acceder en forma privilegiada a aquellos tipos de datos que son representables con mayor facilidad dentro de este encuadre y gozan de una implementación particularmente eficiente, a estos tipos se les conoce como *tipos primitivos* y son los átomos a partir de los cuales se manejará cualquier tipo de información.

Meta

Que el alumno se familiarice con la representación en la computadora de los tipos primitivos de Java.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Visualizar adecuadamente cómo están siendo representados los valores de los tipos primitivos en la memoria de la computadora.
- Identificar el problema de *desbordamiento*[1] al realizar operaciones con números grandes.
- Utilizar apropiadamente la *conversión*[2] de tipos primitivos.
- Preveer los efectos de convertir datos de un tipo a otro tipo.
- Empacar y desempacar bits de información dentro de un tipo con más bits.

[1] En inglés *overflow*.
[2] En inglés *casting*.

ANTECEDENTES

Java tiene ocho tipos primitivos, los cuales se muestran en la Tabla 2.1.

Tabla 2.1 Tipos básicos de Java

Nombre	Tamaño	Representación en memoria
`byte`	1 byte	Entero con signo en complemento a 2
`short`	2 bytes	Entero con signo en complemento a 2
`int`	4 bytes	Entero con signo en complemento a 2
`long`	8 bytes	Entero con signo en complemento a 2
`float`	4 bytes	Racional de acuerdo al estándar IEEE 754-1985
`double`	8 bytes	Racional de acuerdo al estándar IEEE 754-1985
`boolean`	≈1 byte	Booleano `true` o `false`
`char`	2 bytes	Caracter Unicode 2.0

* Un byte son 8 bits

Enteros con signo

Java permite representar números enteros $\mathbb{Z}$ contenidos en intervalos de la forma $[A, B]$ con $A < 0, B > 0$ y $A, B \in \mathbb{Z}$. Los valores de A y B dependen del número de bits disponibles para almacenar estos números y del sistema utilizado para representar a los números negativos. El sistema elegido por `Java` es complemento a dos.

Para calcular la representación de un número negativo utilizando complemento a dos se pueden utilizar tres pasos:

1. Se escribe el número en sistema binario, utilizando tantos bits como indique el tipo entero utilizado. Los bits sobrantes a la izquierda valen cero. El bit más a la izquierda es el bit de signo, en este paso debe valer cero. Por ejemplo, el número 14 en un `byte` se escribe:

   00001110

2. Se calcula el complemento a uno: el valor de cada bit es invertido. Aquí ya aparece el signo menos, indicado por un uno en el bit más a la izquierda.

   11110001

3. Se le suma uno. Ésta es ahora la representación del -14.

   11110010

Existe también una receta corta para pasar de la representación binaria a complemento a dos:

Tabla 2.2 Enteros

Nombre	Tamaño	Intervalo
`byte`	1 byte	$[-128, 127]$
`short`	2 bytes	$[-32768, 32767]$
`int`	4 bytes	$[-2147483648, 2147483647]$
`long`	8 bytes	$[-9223372036854775808, 9223372036854775807]$

1. Se copian, de derecha a izquierda, todos los ceros hasta llegar al primer uno,
2. se copia ese uno y
3. a partir de ahí se invierten todos los demás bits.

Por ejemplo:

$$14_{10} = 00001110_2 \rightarrow 11110010_2 = -14_{10}$$

De acuerdo a esta representación, las capacidades de los diferentes tipos de enteros se muestra en la Tabla 2.2. Puedes acceder a estos valores desde el código de `Java`. Por ejemplo:

```
1 int max = Integer.MAX_VALUE;
```

Actividad 2.1

Revisa la documentación de las clases `Byte`, `Short`, `Integer` y `Long` de `Java` y revisa los atributos de clase que permiten acceder a esta información. ¿Cuáles encuentras relevantes?

En el paquete de código que recibes para esta práctica, crea un archivo `Prueba.java` al lado de `ImpresoraBinario.java`. Dentro escribe un método `main`. Ahí, crea e imprime `max` tanto en base 10 como en binario. Ahora intenta sumarle uno a `max`, imprime el resultado en base 10 y su representación en binario. Explica qué pasó.

Números de punto flotante

Para representar números racionales, `Java` utiliza el estándar IEEE 754-1985. Éste especifica una serie de reglas para utilizar notación exponencial en base dos. Un ejemplo de número binario en notación exponencial sería:

$1.001101 \text{x} 10_2^{-101}$

Tabla 2.3 Flotantes

Nombre	Tamaño	Intervalo
`float`	4 bytes	$[1.4x10^{-45}, 3.4028235x10^{38}]$
`double`	8 bytes	$[4.9x10^{-324}, 1.79x10^{308}]$

Existen buenas referencias para quienes deseen conocer a detalle cómo funciona esta representación [Fish 2005]. De momento lo que nos interesa es lo siguiente:

- Se utiliza el bit más a la izquierda para representar el signo del número: 0 si es positivo y 1 si es negativo.
- Los siguientes X bits se utilizan para almacenar el exponente del dos, en base dos. El primero de esos bits corresponde al signo del exponente.
- Los bits restantes almacenan la mantiza.

La diferencia entre `float` y `double` radica en el número de bits totales y, por ende, los dedicados al exponente y a la mantiza. Los intervalos aproximados de números que pueden ser representados con estos tipos se listan en la Tabla 2.3.

De entre estos números se cuenta con los *normalizados*, cuya mantiza siempre comienza con 1.????. y los no-normalizados, que pueden tener un cero a la izquierda del punto.

Otro aspecto importante de este tipo de datos, es que no sólo representan números fraccionarios, sino también algunos símbolos especiales:

- `NaN`, que quiere decir "*Not a number*". Se utiliza para representar un resultado numérico inválido, como el obtenido al dividir 0/0. Para acceder a este valor en `Java` escribe:

```
1 double nan = Double.NaN;
```

- $\pm\infty$. Para acceder a estos valores en `Java` escribe:

```
1 double minf = Double.NEGATIVE_INFINITY;
2 double pinf = Double.POSITIVE_INFINITY;
```

- ± 0. Como se utiliza la representación signo-maginitud, hay dos ceros. Para acceder a estos valores en `Java` escribe:

```
double cero = 0.0;
double mcero = -0.0;
```

Afortunadamente Java dice que ambos son iguales. Aunque el bit de signo en la computadora es diferente.

Actividad 2.2

Utiliza la clase ImpresoraBinario para visualizar la representación interna de estos valores especiales.

Para ello debes crear una impresora dentro de tu método main, que podrás utilizar a partir de aquí:

```
// Imprime representación de valores especiales
ImpresoraBinario p = new ImpresoraBinario();
p.imprime(nan);
p.imprime(minf);
...
```

Boolean

El tipo boolean es utilizado para operaciones lógicas. Aunque en la lógica booleana sólo hay dos símbolos: $verdadero$ y $falso$, la representación en la computadora ocupa más espacio que un bit. Este espacio se desperdicia, pero se hace para que las operaciones sean más rápidas, debido a que la computadora puede direccionar un byte con mucha mayor facilidad que un único bit.

Cuando se almacenarán muchos de estos valores y no se operará con ellos frecuentemente, es común que el programador *empaque* los valores booleanos en los bits de un tipo numérico. Por ejemplo, los códigos de permisos para lectura, escritura y ejecución de los archivos de Linux utilizan esta técnica. Imaginemos cómo puede lograrse esto:

Hay un total de nueve permisos que manejar, agrupados por tipo de usuario: el *dueño*, su *grupo* de usuarios y *otros* usuarios. Por cada grupo hay tres permisos: *lectura*, *escritura* y *ejecución*. Como cada permiso puede ser *otorgado* o *denegado*, la lógica booleana es adecuada para trabajar con ellos, pero preferiríamos usar sólo un bit por cada permiso, de modo que sólo se requieran nueve bits. Si quisiéramos trabajar en Java, podríamos codificar estos bits dentro de un short.

```
short permisos;
```

Por ejemplo, para decir que un archivo puede ser leído por todos, escrito por el dueño y ejecutado por el dueño y su grupo, necesitaríamos escribir 111101100, donde

los tres primeros bits son los permisos del dueño, los siguientes tres los del grupo y los últimos los de otros. Así quedan registrados `rwx r-x r-`. En base ocho esto es 754, en `Java` se puede asignar:

```
1 permisos = 0754;
```

Observa el cero 0 al inicio del número. Éste le indica a `Java` que estamos escribiendo en base 8. Si no lo pones, creerá que quieres guardar un 754_{10} y los bits no quedarán colocados correctamente.

Actividad 2.3

Dentro del archivo `Prueba.java`, imprime cómo se ve esto en binario. ¿Cuánto vale `permisos` en base 10? (Sólo mándalo imprimir, `Java` trabaja por defecto en base 10).

Operadores sobre bits

Cuando se desea manipular los datos bit a bit, se utilizan los siguientes operadores:

Tabla 2.4 Operadores sobre bits en orden de precedencia.

Operandos	Asociatividad	Símbolo	Descripción
unario prefijo	izquierda	~<expresión>	complemento en bits
binario infijo	izquierda	<<	corrimiento de bits a la izquierda
binario infijo	izquierda	>>	corrimiento de bits a la derecha llenando con ceros
binario infijo	izquierda	>>>	corrimiento de bits a la derecha propagando signo
binario infijo	izquierda	&	AND de bits
binario infijo	izquierda	^	XOR de bits
binario infijo	izquierda	|	OR de bits

Actividad 2.4

Realiza ahora pruebas con los operadores de corrimiento <<, >>, >>>. Recorre los números del iniciso anterior por uno y tres bits. Se usa así:

```
1 int num = 345;
2 int resultado = num << 3;
```

Imprime tus resultados. Asegúrate de que entiendes lo que hizo cada operador.

Actividad 2.5

Ahora toma los permisos de la sección anterior (el valor original de la variable `permisos`): ¿qué operaciones necesitas hacer para que todos los usuarios tengan permiso de escritura, sin modificar sus otros permisos? Verfica tu respuesta imprimiendo las representaciones binarias.

Ejercicios

1. Junto con esta práctica se te entrega el código de la clase `ImpresoraBinario`. En el método `main` de la clase de uso `PruebasPrimitivos` utiliza a `ImpresoraBinario` para imprimir en pantalla la representación con bits de los siguientes números:

   - El `int` 456
   - El mismo número pero en una variable tipo `long`
   - El mismo número pero en una variable tipo `float`
   - El mismo número pero en una variable tipo `double`

   Explica ¿qué diferencias observas?

2. Repite los mismos pasos, pero ahora con -456

3. Repite los mismos pasos, pero con -456.601. Ojo, en este caso deberás hacer un casting para guardar el número en los tipos correspondientes a enteros y perderás la parte fraccionaria. Un casting obliga a `Java` a convertir un tipo primitivo en otro, incluso si se pierde información en el proceso.

```
1 int x = (int)-456.601;
```

   ¿Qué número queda almacenado?

4. Finalmente, crea un `int` llamado `máscara` cuyos últimos cuatro dígitos sean unos. Ahora utiliza el operador de corrimiento necesario, para colocarlos en las posiciones 4 a la 7 (contado de derecha a izquierda). ¿Qué número obtienes?

5. Dado un `int num = 1408`, realiza la operación `num & máscara`. Imprime los bits resultantes y el valor numérico de tu resultado. Repite el ejercicio con `|` y `^`. Calcula `~num`.

3 | Variables tipos y operadores (3)

Leer y realizar los ejercicios de la práctica 3 del manual de Canek y Elisa. No usarás la consola. Pide a tu ayudante que te explique cómo usar `System.out.println`.

NOTA: cambiar la consola por una tortuga.

4 | Programación estructurada

Meta

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

Objetivos

Al finalizar la práctica el alumno será capaz de:

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

Antecedentes

Funciones

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

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

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

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

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

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

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

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

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

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

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

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

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

Polimorfismo: Sobrecarga

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

Ejemplo 4.3 *La función imprime de la clase ImpresoraBinario, que utilizaste en el Capítulo 2, tiene cuatro definiciones distitas, dependiendo del tipo de dato cuyos bits se quieren imprimir en pantalla:*

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

```
public void imprime(int num)
public void imprime(long num)
public void imprime(float num)
public void imprime(double num)
```

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

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

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

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

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

Warnier-Orr

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

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

Para diseñar un algoritmo utilizando este ordenamiento de las instrucciones, se cuenta con la ayuda de la metodología de Warnier-Orr (Viso y Peláez 2012, págs. 52-60). En esta metodología pondremos especial atención en la forma en que recibimos, procesamos y producimos los datos que requiere nuestra función para devolver su resultado. Se analiza el problema sin escribir código, de modo que la solución sea independiente del lenguaje concreto que vayamos a utilizar después. Los símbolos y frases casi asemejan a instrucciones de código, pero van mezcladas con símbolos matemáticos y textos que esbozan los procedimientos a realizar, por ello se le llama *pseudocódigo*.

Secuencial

Cuando los datos se deben procesar en secuencia.

$$\text{nombre} = \begin{cases} \text{descr}_1 \\ \text{descr}_2 \\ \ldots \\ \text{descr}_n \end{cases} \tag{4.1}$$

Condicional

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

$$\text{nombre} = \begin{cases} \text{cond}_1 & \begin{cases} \ldots \\ \ldots \end{cases} \\ \oplus \\ \text{cond}_2 & \begin{cases} \ldots \\ \ldots \end{cases} \\ \oplus \\ \ldots \\ \oplus \\ \text{cond}_n & \begin{cases} \ldots \\ \ldots \end{cases} \end{cases} \tag{4.2}$$

Iterativa

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

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

Recursiva

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

$$nombre = \begin{cases} descr_1 & \text{caso base} \\ f(nombre) & \text{llamada recursiva} \end{cases} \tag{4.4}$$

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

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

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

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

Lo cual solemos desarrollar como:

$$n! = n(n-1)(n-2)...1 \tag{4.7}$$

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

$$\textit{Factorial} = \begin{cases} .Principio & \begin{cases} n \leftarrow \\ \text{¿}n \geqslant 0\text{?} \checkmark \\ f \leftarrow 1 \end{cases} \\ \begin{matrix} fact \\ (n > 1) \end{matrix} & = \begin{cases} f \leftarrow f * n \\ n \leftarrow n - 1 \end{cases} \\ .Final & \begin{cases} \textit{Devuelve } f \end{cases} \end{cases} \tag{4.8}$$

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

Contratos

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

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

Implementación

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

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

Secuencial

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

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

Condicional

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

```
<if>   ::= if (<condición>)
```

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

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

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

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

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

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

Iterativa

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

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

La principal diferencia entre `while` y `do`, es que el primero podría nunca ejecutarse, mientras que está garantizado que el segundo siempre se ejecutará al menos una vez.

En el caso del `for`, `<inicio>` sólo se ejecuta una vez, antes de ejecutar el ciclo por primera vez y `<actualización>` siempre, al final de cada ciclo.

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

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

```
public class Ciclos {
  /**
   * Calcula el factorial de un número.
   * @param n el número cuyo factorial se quiere calcular.
   * @throws ArithmeticException si n < 0.
   */
  public static long factorial(long n) {
    if (n < 0) throw new ArithmeticException
      ("No está definido el factorial de un número negativo");
    long f = 1;
    while(n > 1) {
      f *= n;
      n--;
    }
    return f;
  }

  public static void main(String[] args) {
    long n = Long.parseLong(args[0]);
    long f = factorial(n);
    System.out.println("El factorial de " + n + " es " + f);
  }
}
```

Desarrollo

En esta práctica se modificará la práctica sobre el compilador, del Capítulo 1, para crear un programa que reciba un entero positivo como parámetro e imprima en la pantalla si es un número primo o no.

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

Definición 4.1: Número primo

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

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

$$\text{Nombre del problema} = \begin{cases} .\mathit{Principio} & \begin{cases} n \leftarrow \\ i \leftarrow 2 \\ \mathit{divisible} \leftarrow \text{no} \end{cases} \\ \mathit{divisores} \\ (i <?) & = \begin{cases} i \leftarrow \text{siguiente candidato} \\ \text{es divisible } \{ \\ \oplus \\ \text{no es divisible } \{ \end{cases} \\ .\mathit{Final} & \begin{cases} \text{devolver } \neg \mathit{divisible} \end{cases} \end{cases} \tag{4.9}$$

Ejercicios

1. Cambia el paquete de tu proyecto, para que ahora se llame `icc.funciones`. Modifica los nombres de los directorios y al `build.xml` para que compile tu nuevo proyecto.

2. Modifica el método main para leer el único argumento. Usa la función Integer.parseInt()[2] para obtener el entero.

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

3. Agrega una función estática llamada esPrimo que reciba un entero como argumento y devuelva un boolean indicando si el número es primo o no.

4. Documenta la función que programaste.

5. Manda llamar la función desde el método main, pasando como argumento el número indicado por el usuario.

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

5 | Euclides

Meta

Que el alumno practique el uso de la programación recursiva para resolver un problema sencillo.

Objetivos

Al finalizar la práctica el alumno será capaz de:

1. Plantear un problema en forma recursiva.
2. Identificar el o los casos base en una solución recursiva.
3. Identificar el paso recursivo en una solución recursiva.
4. Programar funciones recursivas en `Java`.

Desarrollo

Esta práctica consiste en resolver un solo problema en forma recursiva, programar la solución en `Java` y ejecutar algunos ejemplos. El problema consiste en implementar el algoritmo de Euclides. **No debes usar ciclos**, todo se puede resolver sólo llamando métodos de `String`.

Para calcular el máximo común divisor de dos números enteros se resta el más pequeño del más grande hasta que queden dos números iguales, que serán el máximo común divisor de los dos números.

Por ejemplo, si comenzando con el par de números 1190 y 476, se tiene:

x	y
1190	476
714	476
238	476
238	238

De donde $\mathrm{m.c.d.}(1190, 476) = 238$

Ejercicios

1. Diseña el algoritmo recursivo para resolver este problema sin programarlo todavía. Escribe la receta.

2. Crea la clase `Euclides` dentro del paquete `icc.recursion`. Recuerda adaptar tu `build.xml` y crear la estructura de directorios correspondiente.

3. Agrega un método que reciba como parámetros dos enteros positivos y devuelva su m.c.d. Ojo, tendrás que revisar precondiciones, deben ser mayores a cero, si no, lanza una `ArithmeticException`.

4. Crea el método main. Si el usuario mandó llamar al programa con dos argumentos (`args.length == 2`), manda a ejecutar tu función con los números que dió el usuario. Si no, ejecuta un demo donde mandes llamar tu función con varios argumentos:

   - Parejas de números pequeños.
   - Parejas de números primos.
   - Parejas de primos relativos (su m.c.d es uno).
   - Que alguno de los números sea 1.
   - Parejas de números no tan pequeños.

   Resuelve a mano los ejemplos que mandaste a ejecutar y asegúrate de que tu programa esté devolviendo los resultados correctos. Haz que tu programa imprima los resultados que calculaste a mano al lado de los que devuelve tu función con algún mensaje de ayuda. Te darás cuenta que se vuelve muy práctico si tienes que corregir algo.

6 | Tipos de Datos y Clases

En esta práctica comenzarás haciendo uso de una clase ya definida en la *Interfaz de Programación de Aplicaciones*, API por sus siglas en inglés[1], de `Java`, para resolver un problema sencillo. En la segunda parte crearás tu propia clase, con atributos y métodos para resolver un problema un poco más complejo.

Meta

Que el alumno identifique las características que debe tener un tipo de dato abstracto, para que otros programadores hagan uso de él. Y que posteriormente comience a definir sus propios tipos de datos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Utilizar objetos de clases definidas por otras personas.
- Definir sus propios tipos de datos para que los utilicen otros programadores.

Antecedentes

Ya viste que un lenguaje de programación incluye un conjunto de tipos de datos básicos, a los cuales se les conoce como tipos de datos primitivos. Los lenguajes orientados a objetos, como `Java`, permiten que definas tus propios tipos de datos a partir de los que ya están definidos. El mecanismo para realizar este trabajo son las *clases* e *interfaces*.

Las interfaces sólo te permiten definir las operaciones de un tipo abstracto de datos,

[1] Application Programming Interface

estrictamente no puedes especificar el conjunto de datos. Las clases, por el contrario, te permite declarar y definir tu nuevo tipo de datos. La correspondiencia es como sigue:

Conjunto de datos: está dado por los atributos de instancia y el conjunto de valores que se le pueden asignar a cada uno.

Operaciones: están definidas con la firma de los métodos. Son funciones cuyo primer argumento son siempre los objetos con los que les manda llamar. Así mismo, puedes utilizar los comentarios de los métodos para establecer contratos con los usuarios de tu tipo de datos, es decir, aquí puedes especificar las precondiciones y poscondiciones que cumplirá tu implementación.

Cuando se aplica una operación sobre un objeto, se dice que se le envía un *mensaje*. El tipo de mensajes que se le pueden enviar a un objeto de un tipo dado, son las operaciones que se pueden hacer con él y están definidas por los métodos de esa clase.

Implementación: Al definir el contenido de los método, es decir, al programarlos, ya les estás dando una forma concreta en la computadora, lo que completa tu tipo de dato.

Cadenas

Para realizar este trabajo deberás utilizar una clase de `Java`: `String`. Esta clase representa secuencias de caracteres (tipo `char`). Cada caracter tiene un índice asociado dependiendo de la posición que ocupa en la cadena. Por ejemplo:

U	n	a	_	c	a	d	e	n	a
0	1	2	3	4	5	6	7	8	9

Dado que se trata de un tipo de datos especial, `Java` se tomó algunas libertades con ella. La forma más sencilla de crear variables tipo `String` es:

```
1 String cad = "Una␣cadena";
```

También puedes concatenar cadenas de dos formas: utilizando el operador + o con un método de la clase:

```
1 String nombre = "Pedro";
2 String saludo = "Hola␣" + nombre;
3 String saludo2 = "Hola␣".concat(nombre);
```

Las cadenas también tienen otra característica especial en Java: no es posible modificarlas. Una vez creadas ya no las puedes cambiar. Pero siempre es posible crear nuevas cadenas como segmentos de otras cadenas, con algún o algunos caracteres diferentes o uniendo cadenas.

Actividad 6.1

Revisa la documentación de la clase `String` en https://docs.oracle.com/javase/10/docs/api/java/lang/String.html. Pon especial atención a los métodos:

- `char charAt(int index)`
- `int indexOf(int ch, int fromIndex)` y similares
- `String replace(char oldChar, char newChar)`
- `String replaceAll(String regex, String replacement)`
- `String substring(int beginIndex, int endIndex)`

Caracteres a cadenas

Ojo, un `char` no es tipo `String`, pero puedes conseguir una cadena con un sólo caracter utilizando la concatenación de la cadena vacía ("") con un caracter:

```
1  String unChar = "" + 'a';
```

Este no es el método más elegante, pero sí el más práctico.

También puedes utilizar los métodos estáticos de la clase `String`:

```
1  String unChar = String.valueOf('a');
```

Observa que en este caso no tienes un objeto de tipo `String` con el cual mandes llamar a la función. Dada la funcionalidad de `valueOf` tiene sentido, pues quieres construir una cadena nueva a partir de un caracter y para ello no deberías necesitar a otra cadena. Este es un buen ejemplo del uso correcto de los métodos estáticos.

Otros símbolos especiales son los caracteres `\n`, que significa *salto de línea* y `\t`, que representa un tabulador horizontal.

Cadenas iguales

Ahora que ya no estás trabajando con tipos primitivos debes tener cuidado. Java trabaja de forma diferente con los tipos primitivos y con las clases. Los valores primitivos pueden ser almacenados en cualquier región de la memoria, mientras que los objetos sólamente pueden residir en una: el montículo o *heap* en inglés.

Cuando declaras una variable de un tipo clase, tu variable no almacena al objeto, sólo tiene la dirección del objeto en el montículo. Así que si comparas dos variables con las direcciones de cadenas diferentes, aunque las cadenas sean iguales, `Java` te dirá que tienen valores diferentes.

```
1 String uno = "Hola";
2 String dos = "Hola";
3 boolean dif = (uno == dos)  // dif es false.
```

Para comparar a las cadenas, en vez de sus direcciones, debes utilizar el método `boolean equals(Object anObject)`.

```
1 dif = uno.equals(dos)       // dif es true.
```

Aprovechando, si quieres ordenar cadenas alfabéticamente puedes utilizar:

- `int compareTo(String anotherString)`.

Actividad 6.2

Revisa la documentación de los métodos de comparación:

- `boolean equals(Object anObject)`
- `boolean equalsIgnoreCase(String anotherString)`
- `int compareTo(String anotherString)`
- `int compareToIgnoreCase(String str)`

¿Qué resultaría de las siguientes comparaciónes? ¿Qué palabra va antes y cuál después?

```
1 "Hola".compareTo("Adios")
2 "Hola".compareTo("hola")
3 "Hola".compareToIgnoreCase("hola")
```

Cadenas a números

Para convertir cadenas a números tendrás que utilizar otros métodos:

- `static short parseShort(String s)`
- `static int parseInt(String s)`
- `static double parseDouble(String s)`

y semejantes para los otros tipos primitivos.

Programando clases

Las *clases* son plantillas que definen tipos de datos. Los *objetos* o *instancias* de la clase son ejemplares concretos. Así como el `5.6` es un ejemplar de `float`, la cadena `"Hola"` es un ejemplar de `String`.

Los tipos de datos que puedes definir están compuestos por otros tipos de datos definidos previamente. De este modo puedes definir el tipo `Círculo`, compuesto por las coordenadas de su centro y su radio.

Ejemplo 6.1 *Los atributos de instancia para cualquier círculo se definen dentro de la clase.*

```
public class Círculo {
  private double cx;
  private double cy;
  private double radio;
}
```

Estos datos solamente deben ser visibles para el círculo y nadie fuera de él, de modo que no puedan ser modificados desde fuera, por ello la declaración de estas variables va precedida por el acceso `private`. Así el círculo mantiene control sobre su estado y sólamente responde a través de *mensajes*. Los mensajes son las operaciones definidas para este tipo de datos y están determinados por los *métodos* de una clase; éstos son funciones que le pertenecen a la clase y operan sobre sus atributos.

Un mensaje escencial para comenzar a trabajar con objetos es el método *constructor*, con el cual se asigna el valor inicial a los atributos del objeto. Este método es diferente pues no indica un tipo de regreso, éste está implícito: devuelve un objeto recién creado, que se comportará según los lineamientos de la clase. El constructor se llama igual que la clase.

Ejemplo 6.2 *Agregamos el constructor y un método para calcular el área del círculo.*

```
package icc.clases;

public class Círculo {
  private double cx;
  private double cy;
  private double radio;

  /** Inicializa un círculo. */
  public Círculo(double cx, double cy, double radio) {
    if (radio < 0) throw new
      IllegalArgumentException("No hay círculos con radio
        ↪ negativo");
```

```
12      this.cx = cx;
13      this.cy = cy;
14      this.radio = radio;
15    }
16
17    /** Calcula el área. */
18    public double calculaÁrea() {
19      return Math.PI * this.radio * radio;
20    }
21  }
```


Observa que el método `área` aparentemente no recibe parámetros. En realidad, todos los métodos de una clase reciben al menos un parámetro: una referencia al objeto sobre el cual debe actuar. Esta referencia se llama `this`. Para evitarnos escribirlo todo el tiempo, `Java` maneja el concepto de alcances: si dentro de un método se intenta usar una variable que no fue declarada en él, entonces asume que se trata de un atributo del objeto `this` y por lo tanto la busca entre las variables declaradas en la clase. Como ejemplo de esto, en el método `área` se invocó la variable `radio` tanto refiriéndose explícitamente al objeto `this`, como omitiéndolo. Ambas versiones funcionan. Ojo, dentro del constructor se muestra que, si hay variables locales que se llamen igual, entonces se vuelve obligatorio usar `this` para distinguirlas.

Para crear ejemplares de una clase se utiliza `new`. Para enviarle un mensaje a un objeto, se utiliza el operador punto.

Ejemplo 6.3 *Creemos un objeto de tipo `Círculo` en el archivo de otra clase. Se requiere que esa clase esté en el mismo paquete que `Círculo`.*

```
1  package icc.clases;
2
3  public class UsoCírculo {
4    public static void main(String[] args) {
5      Círculo c = new Círculo(0, 0, 4);
6      double área = c.calculaÁrea();
7      System.out.println("El área del círculo es " + área);
8    }
9  }
```

Desarrollo

Una mosca parada en la pared

Tu primer ejercicio consistirá en hacer que unas moscas canten la canción "Una mosca parada en la pared". Para que no tengas que escribirla toda, ya está en el archi-

vo `icc.clases.Mosca.java`, que se incluye con esta práctica. Como seguramente recordarás de tu infancia, el chiste de la canción radica en cambiar todas las vocales de la letra original por una sola vocal. Deberás programar el método `public String canta()`, donde la mosca deberá devolver la letra original modificada, según la vocal que se le haya indicado anteriomente con el método `public void setVocal(char vocal)`. Revisa cuidadosamente la documentación indicada anteriormente y lee los comentarios de la clase `Mosca`. Si tienes duda pregunta a tu ayudante. No debe tomarte más de seis líneas resolver este ejercicio utilizando correctamente los métodos de la clase `String`.

RFC

Para la segunda parte de la práctica te toca implementar una clase llamada `Ciudadano`. Como atributos, cada ciudadano tiene:

- un `nombre`,
- un `apellidoPaterno` y
- un `apellidoMaterno` que se almacenan como tipo `String`,
- así como una fecha de nacimiento, que también es una cadena con el formato "dd/mm/aa"

El constructor debe recibir como parámetros estos datos y guardarlos en los atributos correspondiente. Recuerda, nadie debe cambiarle el nombre al ciudadano o su fecha de nacimiento, por lo que esos atributos deben ser `private`.

El RFC se calcula tomando las dos primeras letras del apellido paterno, la incial del apellido materno y la inicial del nombre, seguido de los dígitos finales del año de nacimiento, los dos dígitos del mes y dos dígitos para el día.

Ejercicios

Utilizando los métodos estudiados arriba, resuelve los ejercicios siguientes:

1. Implementa el método `public String canta();` de la clase `Mosca.java`. Al ejecutar `ant run_mosca` deberás ver el texto producido por dos moscas cantando la canción.

2. Implementa la clase `Ciudadano` con su constructor y su método `public String calculaRFC()`. Al ejecutar `ant run_rfc` deberás ver los nombres de tres mexicanos y sus RFCs calculados con tu programa.

3. Agrégale un metodo a tu ciudadano que se llame `String toString()` y devuelva una cadena con el nombre, cumpleaños y RFC del ciudadano. La clase

`UsoCiudadano` lo utilizará para imprimir sus datos en la consola, descomenta el código de su método `main`[2].

4. Recuerda documentar el código de tu clase.

[2]Este código fue comentado para que puedas trabajar la parte de la canción sin tener que implementar `Ciudadano`, de otro modo el compilador te marcará errores.

7 | Contactos

Meta

Que el alumno aprenda a utilizar correctamente las referencias a objetos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

1. Idenficar los efectos colaterales producidos por el acceso a objetos a través de referencias.
2. Aprovechar el uso de referencias para implementar estructuras lineales recursivas de tamaño variable.

Desarrollo

En esta práctica programarás tu propia lista de contactos. Cualquier lista de contactos que se respete debe permitirle al usuario guardar tantos contactos como el usuario desee (mientras haya memoria disponible, claro). Sin embargo ¿cómo podrías guardar 100 registros sin tener que declarar 100 variables? ¿Qué tal sí sólo necesitabas 15? ¿Qué tal si llegas a Presidente de la República y tienes un millón de contactos? Bueno... tal vez eso sea un poco más complicado. Pero el objetivo es poder almacenar un número variable de elementos, sin saber cuántos serán al momento de hacer el programa.

Para lograrlo utilizarás una técnica muy popular que consiste en hacer que cada objeto sepa dónde se encuentra el que sigue y así sucesivamente. Mientras tú tengas la dirección del primer objeto en tu lista, podrás llegar a cualquier otro siguiendo la fila a partir del primero.

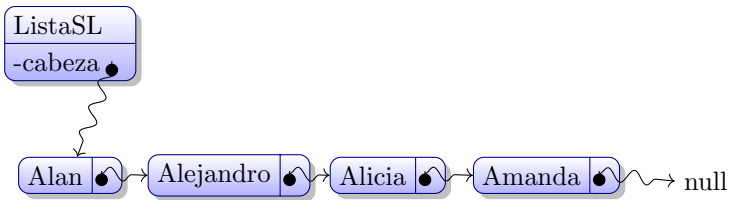

Figura 7.1 Un objeto de tipo `ListaSL` (por *lista simplemente ligada*) tiene una referencia al contenedor con el primer dato. Cada contenedor contiene su dato y la referencia al contenedor con el siguiente dato.

Antecedentes

En `Java`, una forma de programar listas es como una *lista simplemente ligada*, haciendo uso de dos tipos de objetos:

- Una *lista*, que contendrá solamente la dirección de su primer elemento (al cual típicamente se le conoce como `cabeza`), además de los métodos que podrá utilizar el usuario (insertar, consultar, remover, etc.) y
- *contenedores*, que guardarán dentro datos y la dirección del siguiente contenedor. Estos contenedores pueden recibir distintos nombres, pero los más populares son `Nodo` y `Elemento`.

La Figura 7.1 ilustra cómo se visualiza esta esta estructura en la memoria de la computadora.

Ejercicios

1. Crea una clase `Registro`, cuyos atributos serán `nombre`, `dirección` y `teléfono`. ¿De qué tipo debe ser cada uno de esos atributos? Elígelos tú. Recuerda que estas variables deben ser privadas.
2. Ponle un constructor que reciba estos tres datos como parámetros y los asigne a los atributos correspondientes. Asegúrate de que no sean `null`, si el parámetro valiera `null`, asígnales valores por defecto.
3. Agrega *getters* y *setters* para cada atributo, asegúrate de que los valores asignados no sean `null`. Decide cómo se comportará tu programa en caso de que el parámetro valga `null` y documenta tu decisión en el método correspondiente.
4. Agrega otro atributo que será de tipo `Registro` y cuyo nombre será `siguiente`. Agrega su `getter` y `setter` correspondientes, pero no les pongas acceso. Esto

hará que sólo las clases dentro del mismo directorio (`paquete`) puedan mandarlos llamar. Esto permitirá que tu lista de contactos pueda acceder a estos métodos para agregar, consultar, modificar o borrar registros – aunque para esta práctica sólo estarás agregando y consultando. Revisa que tu código compile.

5. Se te da parte del código de la interfaz de usuario en la clase `IUContactos`, completa el contenido del método `solicitaDatosDeContacto()` para que funcione con tu clase `Registro`.

6. Agrega un método `public String toString()` que devuelva una cadena con los datos almacenados en el registro. Ojo: no intentes imprimir en la consola, sólo construye y devuelve la cadena.

7. Crea ahora la clase `Contactos`. Tendrá un atributo de tipo `Registro`, ponle el nombre que quieras pero debe reflejar el hecho de que almacenará la dirección en memoria de tu primer contacto. Al inicio esta variable valdrá `null`, indicando que tu lista de contactos está vacía.

8. Agrega el método `public void insertaContacto(Registro reg)`. Este método debe recibir un objeto de tipo registro con los datos de tu nuevo contacto y lo insertará en orden alfabético. Revisa la documentación del método `compareTo(String a)` de la clase `String`, lo necesitarás para determinar el orden correcto. Aquí es donde tu clase `Contactos` necesitará acceder a los métodos `getSiguiente` y `setSiguiente` de la clase `Registro` para encontrar el registro anterior al nuevo e insertar el registro nuevo en su lugar.

   Para hacer la inserción, deberás reasignar los valores de las variables `siguiente` de los registros involucrados, que son: el registro que insertas, el registro que va antes de éste y el registro que va después y, cuando el registro nuevo se inserte al inicio, también deberás modificar la variable que refiere al primer elemento.

9. Agrega un método `public void imprimeContactos()` que recorra tu lista de contactos y los vaya imprimiendo. Deberán aparecen en orden alfabético. Este es un buen momento para probar tanto la inserción como la impresión de tu lista de contactos. Compila, corre tu código y prueba esas opciones del menú. Si encuentras errores, arréglalos de una vez.

10. Agrega el método `public Registro consultar(String nombre)` que recibe el nombre o una parte del nombre del contacto que quieres consultar. Este método deberá devolver el primer registro que contenga la cadena indicada. Revisa el funcionamiento del método `contains` de la clase `String`, lo necesitarás para resolver esta parte. Nota que deberás revisar todos los registros para ver si alguno contiene la cadena indicada, si ninguno la contiene regresa `null`.

11. Compila y corre tu programa. Ya debe estar funcionando, prueba cada opción del menú y verifica que así sea. Si encuentras errores trata de irlos arreglando opción por opción.

12. Documenta todo tu código y corrige cualquier error de indentación en todas las clases.

Preguntas

1. Al buscar por nombre ¿qué necesitarías hacer para devolver todos los registros que contengan la cadena indicada, en lugar de sólo el primero? Descríbelo, no es necesario que lo programes.

8 | Ajedrez

Meta

Que el alumno se familiarice con el uso de herencia en diseño orientado a objetos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

1. Definir clases y métodos abstractas.
2. Programar clases derivadas de clases abstractas.
3. Dar ejemplos de jerarquías de herencia.

Antecedentes

Diagramas de clase UML

Para expresar mejor el trabajo que se realizará en esta práctica haremos uso de una notación especial: *los diagramas de clase de UML*.

UML son las siglas en inglés de *Unified Markup Language*.

Clases

Las clases se representan con cajas donde se pueden escribir:

- El nombre de la clase.
- Los atributos de la clase.
- Los métodos de la clase.

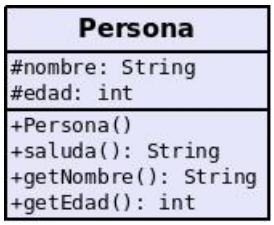

Figura 8.1 Ejemplo de diagrama de clase UML

Para representar los diferentes tipos de acceso se utilizan los símbolos siguientes:

- \- privado. Solamente los métodos de la clase pueden acceder a estos atributos.
- \# protegido. La clase y clases que la extiendan, directa o indirectamente, pueden acceder a estos atributos.
- \+ público. Todos pueden leer y escribir directamente estos atributos. Se recomienda sólo utilizarlos cuando los atributos son `final`, es decir, su valor nunca cambia.

Los accesos también afectan a los métodos e indican quién puede solicitar la ejecución de esos métodos.

Las variables estáticas (o de clase) se destacan por ir subrayadas. Estas son variables compartidas por todas las instancias de una clase. Una imagen dice más que mil palabras, por lo que la Figura 8.1 muestra un ejemplo.

Como verás, la notación para indicar los tipos de las variables es distinta a lo que se utiliza para programar en `Java`. En general, estos diagramas fueron diseñados para ser utilizados con cualquier lenguaje de programación, por lo que su notación es algo independiente de dichos lenguajes y hay ligeras variaciones dependiendo del software que se use para dibujarlos. Aquí seguiremos el estándar implementado por el software para dibujo de diagramas *Dia*. Es software libre y lo puedes descargar gratuitamente. Descubrirás que puede dibujar mucho más que diagramas de clase.

Generalización

La relación de herencia entre dos clases se conoce como una relación de *generalización*, pues se dice que la clase ancestra *generaliza* a sus clases descendientes. En otras palabras, la clase ancestra contiene datos que todas sus descendientes tienen (aún si no los pueden acceder directamente) y puede realizar operaciones que todas sus descendientes pueden realizar.

En UML la generalización se representa con una flecha con un triángulo vacío como punta, desde la clase que extiende hacia la clase más general [Figura 8.2].

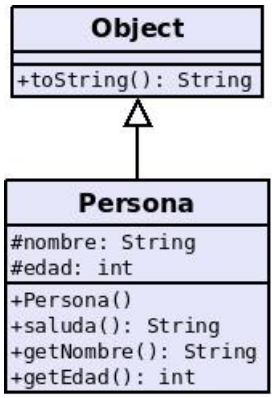

Figura 8.2 Ejemplo de generalización UML

Figura 8.3 Ejemplo de una interfaz en UML

Interfaces

En su versión más simple, las interfaces únicamente contienen declaraciones de métodos y no atributos. No todos los lenguajes de programación tienen interfaces como `Java`, por lo que es necesario construir un símbolo para ellas a partir de las opciones que ofrece UML. Una opción común se ilustra en la Figura 8.3.

Realización

Curiosamente, UML sí tiene un símbolo para indicar que una clase cumple con un contrato. Cuando una clase implementa una interfaz utilizaremos una flecha semejante a la de herencia, pero con la línea punteada. La Figura 8.4 muestra cómo dos clases implementan una interfaz.

Contención

Las relaciones de *contención* indican que un objeto contiene a otro (como un atributo). Se distinguen dos tipos especiales:

Agregación El objeto contenido puede existir, aunque el contenedor deje de existir. Se representa con un diamante vacío del lado de la clase contendora. Por ejemplo: un `Chofer` existe aún afuera de su coche, el `Coche` sigue existiendo aunque

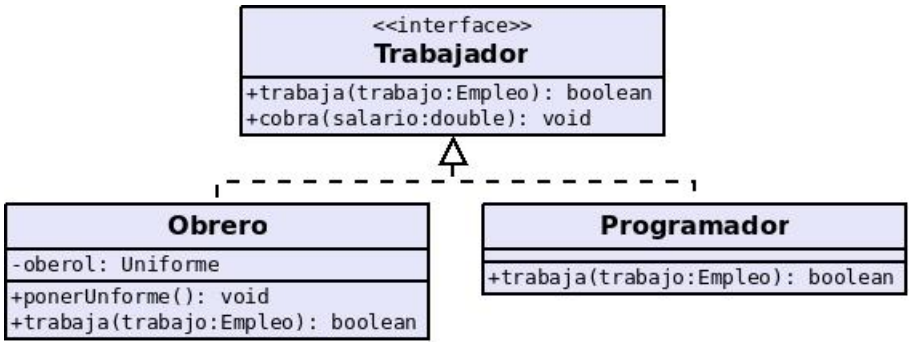

Figura 8.4 Ejemplo de realización UML

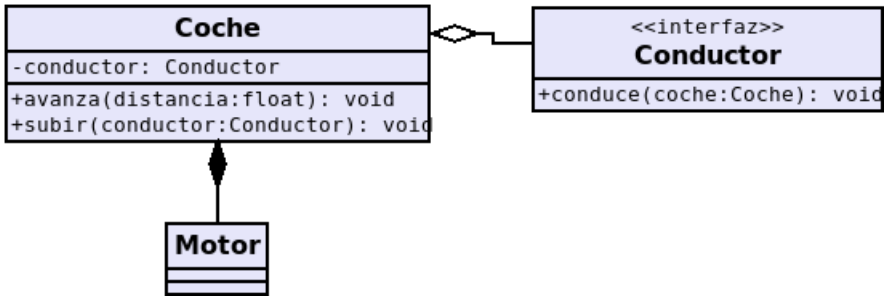

Figura 8.5 Ejemplos de contención UML. Agregación entre `Coche` y `Conductor`. Composición entre `Coche` y `Motor`.

se haya salido su chofer. Pero cuando esté manejando el coche, el chofer estará dentro de él Figura 8.5.

Composición El objeto contenedor está hecho de los objetos que contiene. Se representa con un diamante lleno del lado de la clase contendora. El ejemplo más tradicional son el cuerpo y las partes del cuerpo (brazos, piernas, torso, cabeza). También el coche está hecho de motor, carrocería, puertas, etc. y si falta alguno de estos componentes ya no tenemos un coche completo.

Dependencia

La relación de *dependencia* indica que una clase depende del trabajo realizado por otra (por ejemplo, de la respuesta que calcule cuando mande llamar a un método suyo) o de la estructura que define. En su forma más general, esta relación se representa con una flecha cuya punta está dibujada con dos líneas, desde la clase que ocupa el servicio hacia la que lo proveé. De hecho, la generalización y la realización son tipos particulares de relaciones de dependencia, que tienen asignada su propia variante en la notación.

Asociación

Cuando dos clases están relacionadas de cualquier forma, se dice que están asociadas. La forma más genérica de representar una asociación es con una línea uniendo ambas clases.

Paquetes

Por último, si queremos ilustrar las relaciones entre las clases de un sistema complejo, querremos indicar la agrupación de sus componentes en paquetes con funcionalidades distintas. El símbolo para los paquetes es un folder con el nombre del paquete en su pestaña.

El estándar de UML contiene muchos más símbolos y formas de utilizar la notación, que resultan útiles durante el proceso de ingeniería del software; sin embargo estos serán suficientes para que trabajemos ahora. Observa también que la cantidad de detalles que se incluyen en el diagrama puede facilitar o dificultar su lectura. Cuando se utilizan para documentar un proyecto es importante buscar un buen balance entre la expresividad[1] del diagrama y la claridad de lo que ilustra. Algunos paquetes de software también permiten generar cógido a partir de estos diagramas y, para que el código sea correcto, es necesario seguir las restricciones impuestas por ese paquete.

Diseño orientado a objetos

Todo sistema programado con el paradigma orientado a objetos busca promover las siguientes cualidades Viso y Peláez 2007:

Modularidad Se deben trazar fronteras claras entre partes del sistema para que la tarea se pueda repartir entre varios programadores.

Bajo nivel de acoplamiento Cada módulo debe usar lo menos posible de otros módulos, de forma que puedan ser reutilizados en otras aplicaciones.

Alta cohesión Todos los elementos que se encuentren altamente relacionados entre sí, deben encontrarse dentro del mismo módulo. Esto hará que sean más fáciles de localizar, entender y modificar.

[1] Es decir, cuánta información proveé.

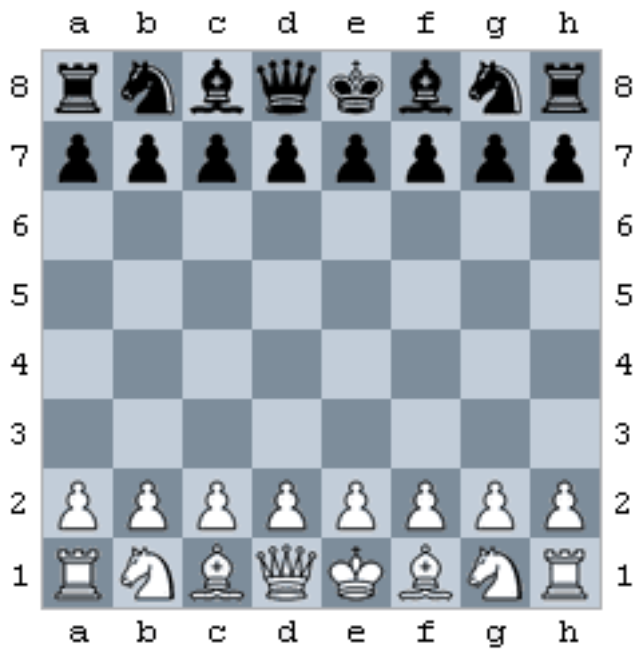

Figura 8.6 Etiquetado de las casillas en un tablero de ajedrez.

Desarrollo

En esta práctica programarás los algoritmos para calcular las casillas a las cuales se pueden mover las piezas de un juego de ajedrez, sin considerar los casos cuando capturarán una pieza enemiga[2]. Para ello, la Figura 8.6 muestra el lenguaje estándar para indicar la posición de una pieza en el tablero de ajedrez. Para los fines de esta práctica, no necesitarás modelar el tablero, sólo las piezas. Por otro lado, la Figura 8.8 muestra los movimientos válidos para cada pieza.

Ejercicios

Trabajarás en un paquete llamado `icc.ajedrez`. Copia un archivo `build.xml` de alguna práctica anterior y modifícalo para que funcione con los archivos de esta práctica. Observa que tendrás más de una clase con método `main` por lo que necesitarás agregar los comandos correspondientes. Escribe en un archivo `.txt` las instrucciones para utilizar tu programa.

1. Implementa una pequeña clase `Posicion` que contenga las dos cantidades: renglón y columna. Agrega los *getters* y *setters* que consideres necesarios. Ase-

[2]Esta práctica está inspirada en 11.1.Ajedrez (López Gaona 2012, pág. 100)

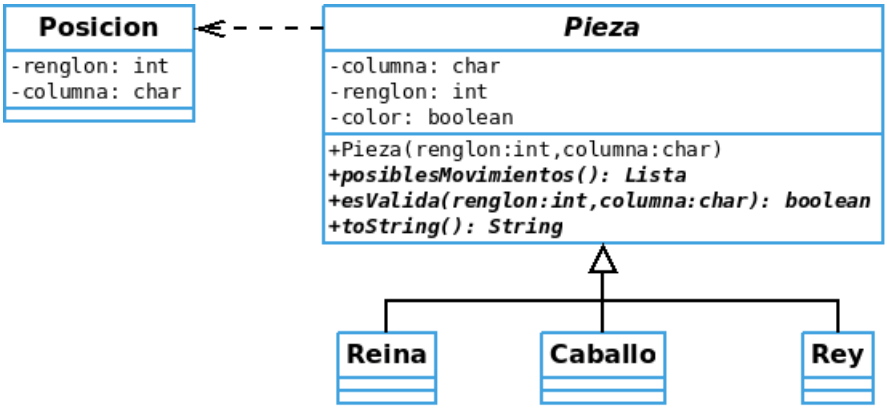

Figura 8.7 Diagrama de clases para las piezas de ajedrez.

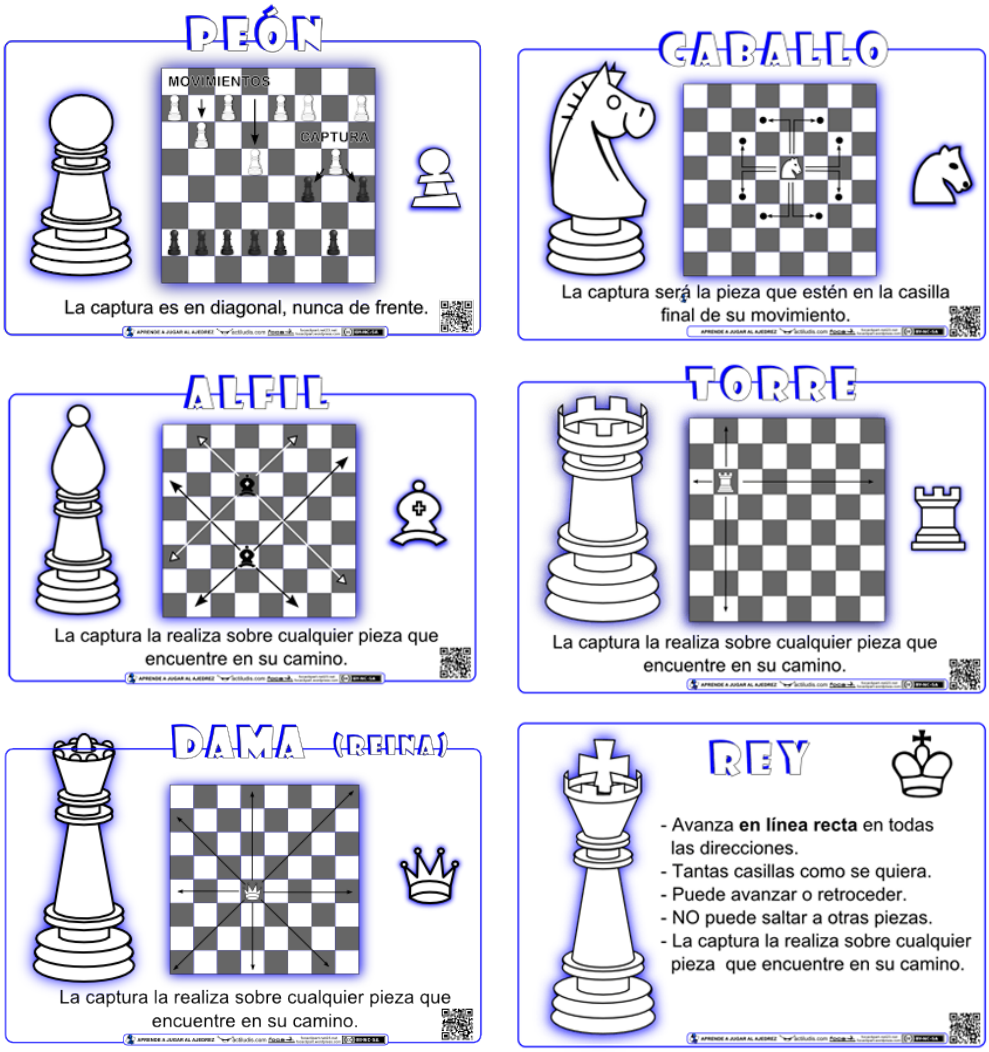

Figura 8.8 Movimientos de las piezas de ajedrez.

gúrate de que, cada vez que se asignen valores a `renglon` o `columna`, sólamente se acepten valores válidos. Si los valores de entrada no lo son, lanza una `IllegalArgumentException`. Es decir que, para renglón, los valores deben ser enteros entre 1 y 8, mientras que las columnas son los caracteres $\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}, \mathrm{g}, \mathrm{h}$.

Crea una clase llamada `PruebaPosicion`, con un método `main` donde crees objetos de tipo `Posicion`. Intenta asignarle valores válidos e inválidos. Verifica que su comportamiento sea el correcto. Para los casos en que se debe lanzar una excepción, comenta la línea de código correspondiente para realizar la siguiente prueba. Cuando entregues tu código déjalas ahí, pero comentadas.

2. Implementa la clase `Pieza`, debe ser abstracta pues incluirá tres métodos abstractos:

   - `posiblesMovimientos()` Dada la posición actual de la pieza, devuelve una `Lista` con las posiciones de todas las casillas a las cuales se podría mover esa pieza. Aquí es donde la clase `Posicion` resultará particularmente útil.
   - `esValida(int renglon, char columna)` Indica si sería válido mover a la pieza desde su posición actual hasta la posición indicada en los parámetros. Debe tomar en cuenta, en particular, que la pieza no se salga del tablero.
   - `toString()` Devuelve una representación con cadena de la pieza y su estado actual.

   Agrega los *getters* y *setters* que consideres necesarios.

3. Programa la clase `Reina`. Según las reglas del juego puede desplazarse a cualquiera de las casillas en línea recta, horizontal o diagonal a partir de su posición, tanto hacia adelante como hacia atrás. En el mejor de los casos hay hasta 32 posiciones posibles.

4. Programa la clase `Caballo`. El caballo se puede desplazar en forma de L: dos casillas verticalmente y una horizontal o a la inversa. Puede tener hasta 8 posiciones a las cuales moverse.

5. Programa la clase `Rey`. El rey puede moverse una sola casilla en las ocho direcciones, por lo que hay máximo ocho posiciones.

   Para completar el programa, habría que agregar `Torre`, `Alfil` y `Peon`, pero sólo te serán útiles si de verdad quieres programar el juego y para eso aún falta ver cómo programar el tablero y determinar las capturas. De momento no haremos eso.

6. Agrega una clase de uso, `UsoAjedrez`, donde crees una reina, un caballo y una torre. Pide que genere las posiciones posibles a partir de celdas en el centro

y orilla del tablero; así como que indique si la posición es válida a partir de la posición actual tanto para propuestas válidas, propuestas que no estén sobre una casilla válida y posiciones fuera del tablero (ej $[-2, \mathrm{i}]$). Observa que, para cambiar la casilla actual de las piezas, deberás haber programado los *setters* correspondientes y estos deben impedir que se asignen posiciones fuera del tablero.

7. Se darán dos puntos extra si programas una interfaz de usuario que permitan realizar lo anterior mediante un menú para el usuario. Se otorgarán fracciones de estos puntos si la interfaz permite realizar sólo algunas de estas tareas.

9 | Listas y manejo avanzado de objetos

Una vez que se conocen las construcciones de un lenguaje de programación para controlar el flujo de ejecución del código, es posible manejar cantidades de información cuyo tamaño se desconoce al momento de codificar el programa. En esta práctica se hará uso de listas para contar con un almacén de datos de tamaño variable.

Antecedentes

Listas

Las listas son un tipo abstracto de datos que nos permite tener un número arbitrario de elementos de un tipo dado. Formalmente, la podemos definir como:

Definición 9.1

Una *lista* es:

- Una lista vacía.
- Un elemento seguido de otra lista.

La definición de una lista es *recursiva*: se define en términos de sí misma. En este caso decimos que se trata de una *recursión estructural*, pues las listas son objetos (estructuras) que contienen objetos del mismo tipo que ellos.

Toda recursión contiene en su definición un *caso base*, es decir, un caso muy sencillo a partir del cual se construyen los más complejos. En este caso el caso base es cuando la lista es una lista vacía, pues ésta ya no contiene a otra lista. Puedes visualizarla así:

$$l = \{\}$$

A partir de ahí se construyen las listas más largas, si los elementos son letras, otros ejemplos de listas son:

$$l_1 = \{A\{\}\}$$
$$l_2 = \{L\{I\{S\{T\{A\{\}\}\}\}\}\}$$

Observa que cada lista contiene a un elemento y otra lista dentro, al final siempre queda un lista vacía.

En `Java` usaremos a `null` para representar a lista vacía. Si nuestra lista contiene objetos de tipo cadena, podemos definir a la clase con los atributos siguientes:

Listing 9.1: Lista.java

```
1 public class Lista {
2   private String elemento;
3   private Lista siguiente;
4 }
```

Los métodos que operan sobre las listas tendrán que trabajar de forma distinta dependiendo de si ésta es vacía o no. En orientación a objetos nos gusta que este tipo de detalles queden ocultos al programador. Después de todo, una lista es una lista, querríamos tener un objeto que agregue elementos a lista y no nos distraiga con este tipo de detalles. Por ello utilizaremos un `ManejadorDeLista`. Éste se encargará de resolver todos los detalles referentes a la administración de la lista y nos dejará preocuparnos únicamente por agregar, buscar y, en general, operar con nuestros datos, independientemente de cómo los esté guardando.

Método toString()

Toda clase de `Java` hereda de la clase `Object` ya sea directa o indirectamente. La clase `Object` tiene un método llamado `toString()` cuyo trabajo es devolver una cadena que represente al objeto y todas las clases heredan este método. Por defecto, esta cadena lista información como la clase del objeto y la dirección de memoria donde se encuentra. Pero si quieres una representación más amigable para los objetos de las clases que programes, lo que debes hacer es *sobreescribir* este método, es decir, declarar un método en tu clase que tenga la misma *firma*. En el código de ese método devuelve la cadena que quieres que represente a tu objeto. Ahora cada vez que `Java` llame al método `toString()` para tu objeto, verás la cadena que tú devolviste. El ejemplo más inmediato es cuando uses `System.out.println()` pues por dentro manda llamar a este método.

```
1 public class UsoListaRegistro {
2   public static void main(String[] args) {
3     Rosa rosa = new Rosa();
4     System.out.println(rosa);
5   }
6 }
```

Actividad 9.1

Lee la documentación del método `System.out.println`. ¿De qué clase es instancia el objeto `out`? ¿Cuántas veces está sobrecargado el método `println`, da algunos ejemplos? https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#out.

Actividad 9.2

Por un punto extra en esta práctica explica ¿por qué hay una versión de `println` por cada tipo primitivo de `Java`?

EJERCICIOS

Para esta práctica se te entrega un archivo comprimido con dos paquetes, con varios archivos:

- `icc.agenda`
  - ⋆ `UsoBaseDeDatosAgenda.java`
  - ⋆ `BaseDeDatosAgenda.java`
  - ⋆ `RegistroAgenda.java`
  - ⋆ `BuscadorPorNombre.java`
- `icc.util`
  - ⋆ `ManejadorDeLista.java`
  - ⋆ `ListaAgenda.java`
  - ⋆ `IBuscador.java` - Este archivo es en realidad una interfaz.

La arquitectura de la pequeña pieza de software que estás programando se muestra en Figura 9.1.

1. Abre el archivo `ListaAgenda.java`, el trabajo de este objeto es almacenar los datos de la lista. Concretamente, los datos que almacenará serán de tipo `RegistroAgenda`. Completa el código faltante en esta clase. Compila tu código y asegúrate de que no tenga errores.

2. Después de compilar tu código podrás solicitar a `ant` que genere la documentación de las clases que se te dieron utilizando el comando `ant docs`. Lee la documentación de la clase `ManejadorDeLista`.

3. Abre `BaseDeDatosAgenda.java`. Completa el código que falta dentro del constructor. Debes extraer los datos de cada registro de la cadena `datos`, que recibe como parámetro. Para cada registro crea un objeto `RegistroAgenda` y guárdalo dentro de la lista utilizando al `ManejadorDeLista`.

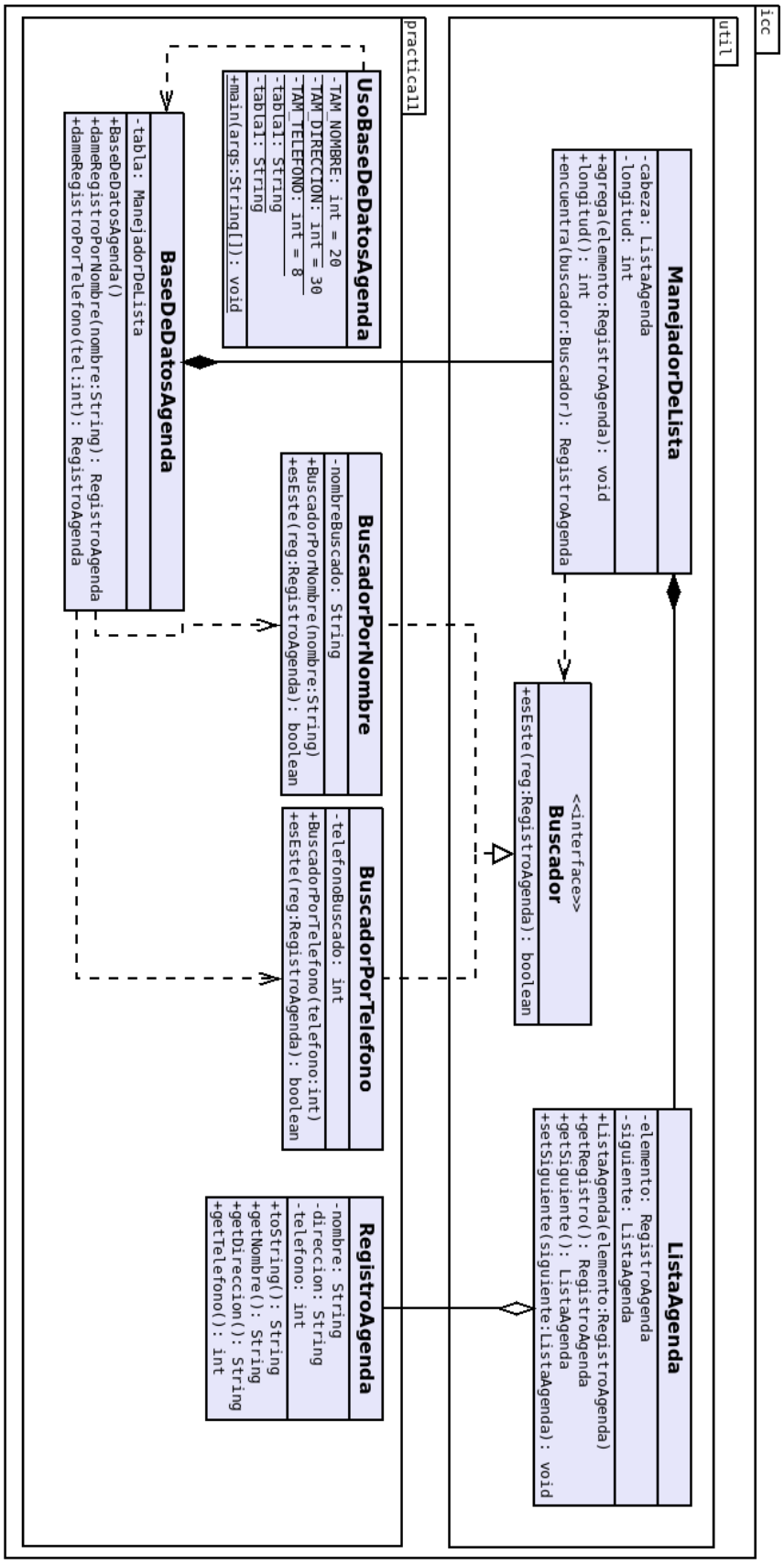

Figura 9.1 Diagrama de clases UML de un pequeño sistema.

4. Prueba tu código creando un par de `BaseDeDatosAgenda` en el método `main` de `UsoBaseDeDatosAgenda`.

5. En la clase `ManejadorDeLista` agrega un método llamado `imprime` que no devuelva ningún valor, ni reciba ningún dato, e imprima en `System.out` el contenido de la lista. Crea un método con la misma firma en `BaseDeDatosAgenda`, que mande llamar al método `imprime` con el objeto `tabla`. Ahora manda llamar ese método con las dos bases que creaste en el `main` de `UsoBaseDeDatosAgenda`. Vuelve a ejecutar el programa, deberás ver en pantalla los datos que agregaste.

6. Ahora comenzaremos con las búsquedas. Lee la documentación de la clase `BuscadorPorNombre` y `IBuscador`. Asegúrate de que entiendes lo que hace cada método. ¿Te queda clara la relación entre la interfaz `IBuscador` y la clase `BuscadorPorNombre`? Si no, pregunta a tu ayudante.

   Supon que `ManejadorDeLista` está completa, revisa la documentación de su método `encuentra(IBuscador buscador)`. Trata de escribir el código del método `dameRegistroPorNombre (String nombre)` de `BaseDeDatosAgenda`. Mándalo llamar en el método main de `UsoBaseDeDatosAgenda` haciendo una consulta. Compila y trata de ejecutar tu código. No va a funcionar, porque `encuentra(IBuscador buscador)` ahorita no hace nada. Pero observa que ya tienes lista la prueba para tu código, aunque no esté completo. Esto es importante porque así podrás verificar que esté bien cuando lo termines.

7. Abre el archivo `ManejadorDeLista.java`, observa que el primer elemento de la lista (si no es vacía) debe quedar en el atributo `cabeza`. Deberás completar el código del método `encuentra(IBuscador buscador)`, que devuelve al registro que cumple la condición que revisa `IBuscador`. Puedes leer cómo fue programado `agrega(RegistroAgenda elemento)` para darte algunas ideas. Cuando termines compila y ejecuta tu código. La búsqueda por nombre ya debe funcionar.

   Asegúrate de probar que puedes encontrar a la última persona en la agenda y que tu programa no hace nada raro si buscas a alguien que no está.

8. Ahora sólo agrega la búsqueda por teléfono. También debes probarla.

10 | Refactorización

Meta

Que el alumno modifique su programa para aplicar el principio de *acoplamiento mínimo*, separando componentes que realizan trabajos independientes.

Objetivos

Al finalizar la práctica el alumno será capaz de:

1. Implementar una lista que almacene cualquier tipo de objetos.
2. Utilizar dicha lista para una aplicación particular.
3. Adaptar un programa diseñado originalmente para un objetivo particular, de tal modo que sus componentes se vuelvan más independientes y, por lo tanto, más reutilizables.

Antecedentes

Contención

Las relaciones de *contención*[1] en esta práctica son particularmente interesantes.

Agregación Un `RegistroAgenda` puede existir, con los datos de la persona registrada, aunque ya no tengamos ni la lista ni la base de datos; pero comunmente está contenido en la lista que lo almacena.

Composición La Figura 10.1 muestra el caso de `ManejadorDeLista` y `ListaAgenda`. `ListaAgenda` no puede existir fuera de su manejador, nadie más la ve ni la usa.

[1] Figura 8.5.

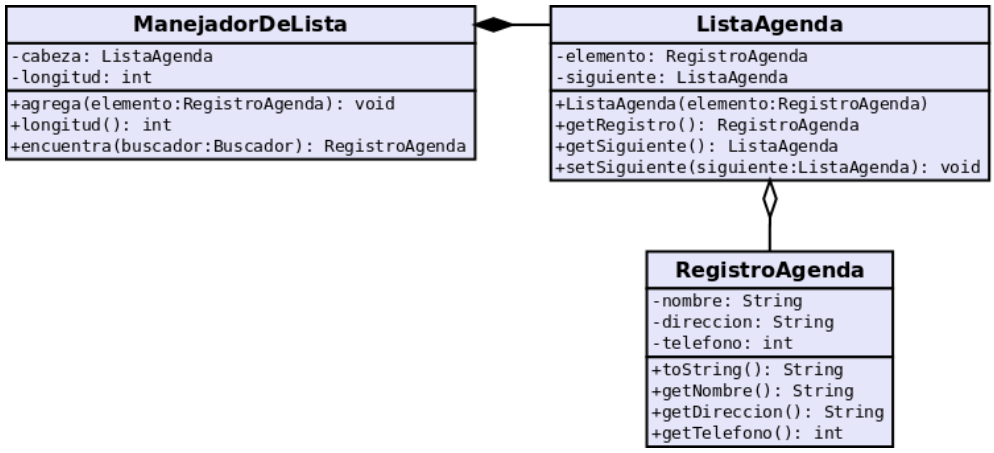

Figura 10.1 Ejemplos de contención UML. Agregación entre `ListaAgenda` y `RegistroAgenda`. Composición entre `ManejadorDeLista` y `ListaAgenda`.

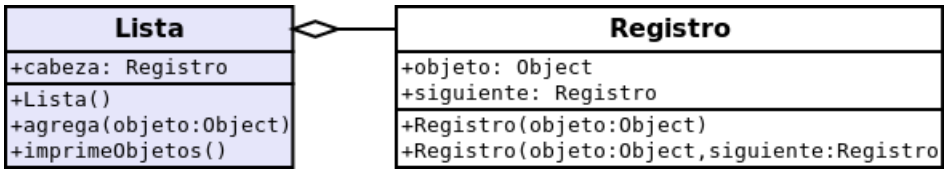

Figura 10.2 Diagrama UML que muestra la relación entre la clase `Planta`, su `Especie` y las clases correspondientes a tipos concretos de plantas.

Listas de Objetos

Al programar tu lista de contactos ya aprendiste a programas listas. Tienes dos clases: el registro, que contiene los datos y la dirección del siguiente registro en la lista, y una clase que se encarga de manipular a toda la lista. Ésta última encapsula la forma en que almacenaste los datos en la memoria de la computadora y permite que el usuario de la lista se imagine que los datos se encuentran almacenados secuencialmente.

Para esta práctica repetirás la misma estructura, pero aprovecharás la herencia para que, a partir de ahora, puedas almacenar cualquier objeto en la lista que programes. La idea es que no tengas que repetir el mismo código cada vez que quieras guardar algo diferente en los registros. Para ello, en lugar de nombre, dirección y teléfono, lo que irá dentro de un registro será un `Object`. Dado que todas las clases en `Java` heredan de `Object`, podrás guardar objetos de cualquier tipo en la lista. La Figura 10.2 muestra esta estructura.

A continuación se incluye un extracto de código que muestra cómo se deben utilizar estas listas:

```java
public class UsoListaRegistro {
  public static void main(String[] args) {
    Lista rosas = new Lista();
    rosas.agrega(new Rosa()); // Agrega el objeto al final
    rosas.agrega(new Rosa());
    rosas.imprimeObjetos();
  }
}
```

Desarrollo

Arquitectura del programa

La Figura 10.3 muestra la estructura de las clases tal y como quedaron en la práctica anterior e ilustra el uso de los componentes descritos anteriormente.

Diseño orientado a objetos

El diagrama anterior muestra algo muy extraño en nuestro sistema:

1. La lista de los registros agenda y sus clases relacionadas se encuentran en otro paquete. ¡Pero `RegistroAgenda` esta en `icc.agenda`!

Actividad 10.1

Escribe, antes de continuar leyendo, tu hipótesis sobre porqué tiene esta estructura el sistema de la práctica. (No, no fue un accidente.) Toma en cuenta que ahora sabes de herencia.

Ejercicios

Para esta práctica utilizarás el trabajo que realizaste en la práctica anterior, por lo que necesitarás los archivos siguientes:

- `icc.agenda`
  - ⋆ `UsoBaseDeDatosAgenda.java`
  - ⋆ `BaseDeDatosAgenda.java`
  - ⋆ `RegistroAgenda.java`
  - ⋆ `BuscadorPorNombre.java`

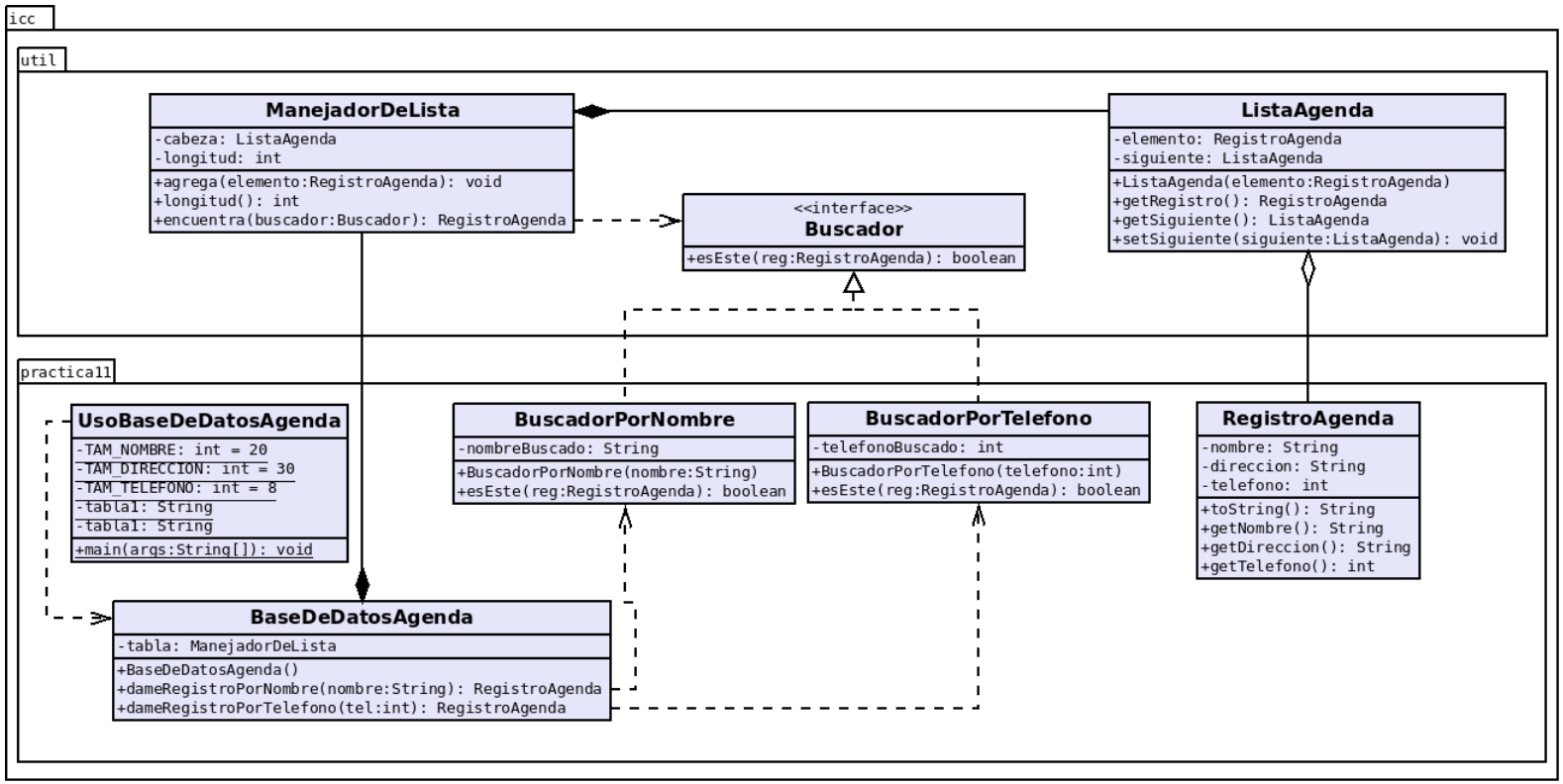

Figura 10.3 Diagrama de clases UML de un pequeño sistema.

    - ⋆ BuscadorPorTelefono.java
- icc.util
    - ⋆ ManejadorDeLista.java
    - ⋆ ListaAgenda.java
    - ⋆ Buscador.java

En esta práctica vamos a *refactorizar* el código de la práctica anterior. Este es un proceso por el cual pasan los ingenieros de software conforme su sistema crece y su código debe cumplir nuevas funciones, aún más generales que las originales. ¿Recuerdas lo que significa *factorizar* en álgebra? Cito un ejemplo a continuación para refrescar la memoria:

$$ab^3x^2 + 2ab^3xy + ab^3y^2 = ab^3(x^2 + 2xy + y^2) \tag{10.1}$$
$$= ab^3(x + y)^2 \tag{10.2}$$

Como recordarás, para factorizar un polinomio, debes buscar los elementos comúnes a todos los monomios y *extraerlos*, para luego realizar otras simplificaciones. Pues bien, con el código se hace lo mismo, pero a nivel conceptual.

El código se encuentra a la mitad de uno de estos procesos. Las clases de la base de datos están divididas en dos paquetes para dar una pista. En este sistema hay dos funcionalidades completamente diferentes:

1. Crear y manipular una lista, que se usa para contener y consultar secuencias de datos.

2. Manejar la agenda.

Ambas cosas son distintas: la lista no necesita a la agenda (conceptualmente), y la agenda podría preferir guardar sus datos en algún otro lado, que no sea la lista. Así que ambas cosas deberían ser separables manteniendo únicamente la comunicación necesaria para que la agenda guarde sus datos en la lista y la use para hacer consultas.

Bueno, esto no se podía hacer mientras no tuviéramos herencia. Para que una lista pueda almacenar cualquier cosa, haremos uso de un hecho básico en `Java`: todas las clases heredan de la clase `Object`. Así que, donde podamos poner un objeto, podemos poner un `RegistroAgenda`. Observa que las clases que se encuentran en el paquete `icc.util` no necesitan ninguno de los métodos de `RegistroAgenda` y podrían hacer exactamente el mismo trabajo si recibieran `Object` en lugar de `RegistroAgenda`. Por ello es que fueron puestas dentro de un paquete diferente; las podemos separa de las demás.

Por otro lado, si las listas guardan objetos, la base de datos tendrá que lidiar con algunos `castings`. Cuando la lista devuelva un `Object`, las clases en `agenda` deberán recordarle al compilador que se trataba de un `RegistroAgenda`. Esto se vería como:

```
1  RegistroAgenda rg = (RegistroAgenda)(tabla.encuentra(buscador));
```

De ahí en fuera, el código es el mismo.

1. Tu tarea para esta práctica es modificar las clases para que cumplan ahora con lo especificado en la Figura 10.4.
2. Arregla primero lo que esté en `icc.util`. Sólo debes cambiar tipos, no debes necesitar cambiar otro código.
3. Ahora ajusta `icc.agenda` para que funcione con las modificaciones, cambia los nombres de los constructures, agrega castings y cualquier otro pequeño detalle que impida que funcione tu código. Verifica que tu programa vuelva a funcionar como antes.

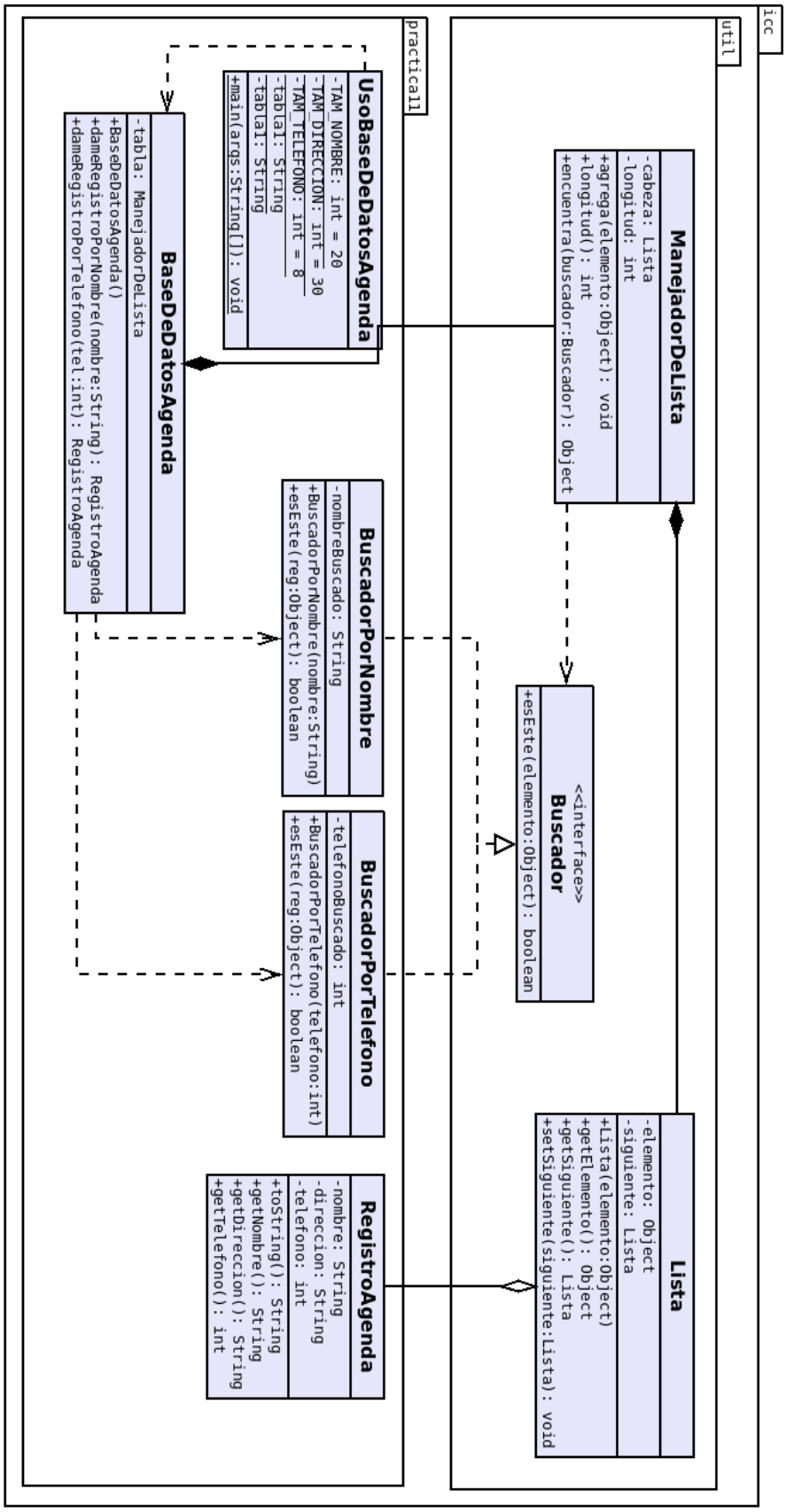

Figura 10.4 Diagrama de clases UML de un pequeño sistema.

11 | Base de datos persistente

Hasta ahora, todo el trabajo que has hecho se mantiene en la memoria de la computadora mientras tu programa esté corriendo. Cada vez que abres tu lista de contactos esta vacía y debes agregarlos a mano, pero cuando termina su ejecución desaparecen los contactos que tanto trabajo te costó agregar. Lo que harás en esta práctica es almacenar los registros de tu `BaseDeDatosAgenda` para que la próxima vez que la ejecutes, tu información siga ahí.

Meta

Que el alumno aprenda a escribir y leer información del disco duro.

Entrada y salida en Java

Definición 11.1: Flujo

Un **flujo** es una secuencia de datos. Un programa utiliza un **flujo de entrada** para leer datos desde una fuente, uno a la vez, y un **flujo de salida** para enviar datos a un destino, también uno a la vez. *I/O Streams* 2016

`Java` distingue entre dos formas de manejar esa información: binaria y como caracteres. Un grupo de clases lee y escribe bytes, mientras que otro está preparado para trabajar con diferentes codificaciones de texto. Las clases más comunmente utilizadas de ambos paquetes están ilustradas en Figura 11.1.

Para esta práctica necesitarás estar consultando constantemente la documentación de la API de `Java`, así que ten la dirección a la mano: `http://docs.oracle.com/javase/8/docs/api/`.

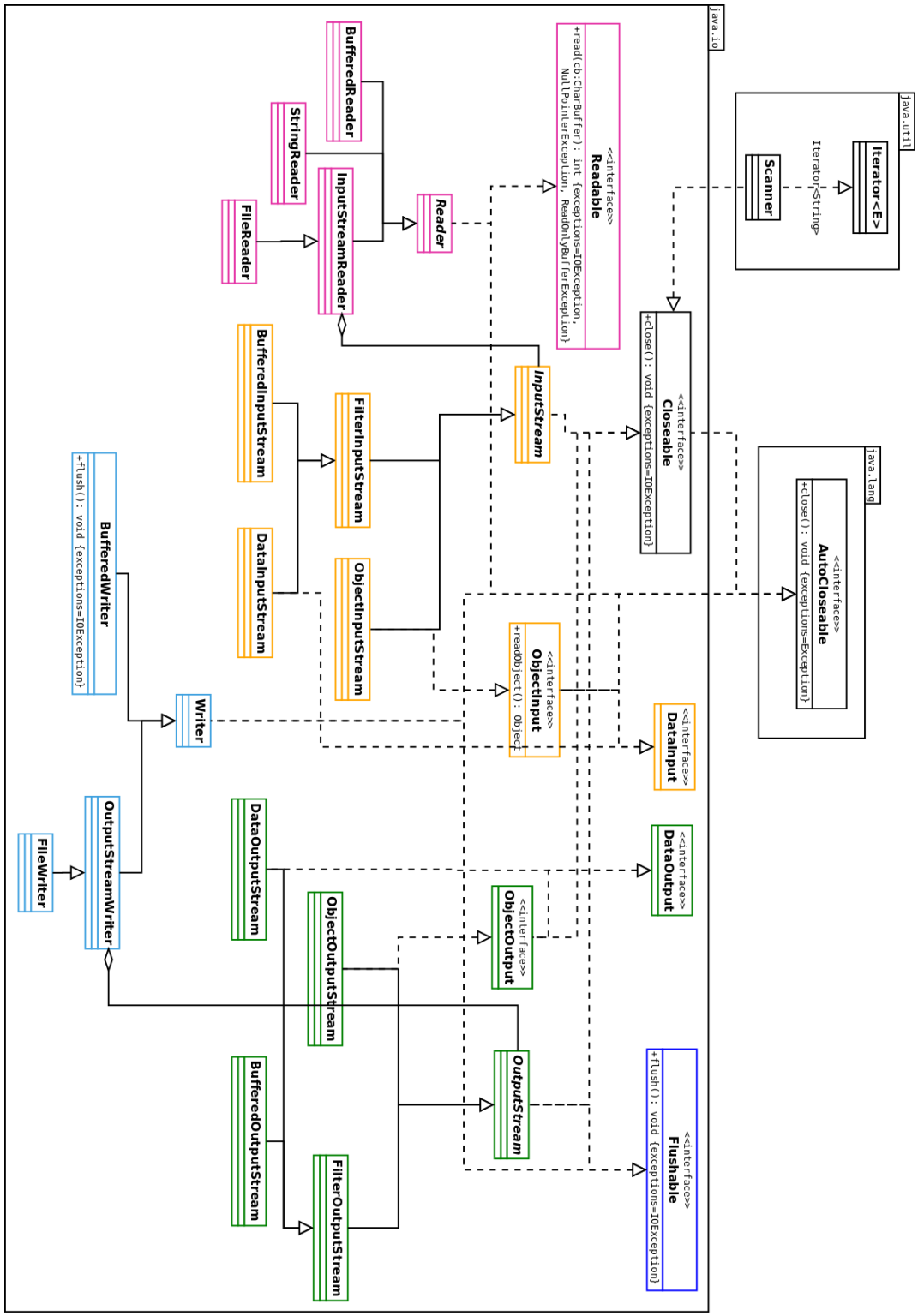

Figura 11.1 Diagrama de clases UML de algunas clases para entrada/salida en Java.

Recibiendo entrada desde la terminal

Si quieres hacer un programa con interfaz de usuario basada en texto, como el `bash` que se ejecuta en la terminal, necesitas acceder al flujo de entrada de la terminal. `Java` hace esto a través del objeto estático `System.in`. Éste es un `InputStream` y lee bytes. No queremos tener que decodificar cada caracter que teclee nuestro usuario byte por byte. El tipo de objeto que puede ayudarnos con ese trabajo es un `Scanner`. Si al `Scanner` le damos un flujo de entrada cuando lo construímos, él se encargará de hacer la decodificación por nosotros y devolvernos cadenas de caracteres (`String`). A continuación se incluye un pequeño ejemplo de cómo utilizarlo para este fin.

Actividad 11.1

Revisa la documentación oficial de `java.util.Scanner`. ¿Qué es un objeto tipo `File`?

```java
import java.util.Scanner;

/**
 * Demo básico sobre el uso de la clase Scanner.
 * @author blackzafiro
 */
public class DemoScanner {

  /**
    * Recibe una línea de texto desde el teclado y
    * reimprime el mensaje que recibió, hasta que el usuario
    * se despida.
    * @param args
    */
  public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    while(s.hasNext()) {
      String linea = s.nextLine();
      System.out.println("Eco: " + linea);
      if(linea.equals("Adios")) break;
    }
    s.close();
  }

}
```

Actividad 11.2

Copia y ejecuta el código anterior. Para hacerlo más rápido que con `ant`, cópialo en un archivo `DemoScanner.java` e invoca al compilador escribiendo `javac DemoScanner.java`. El archivo `.class` se generará ahí mismo. Ejecútalo con

`java DemoScanner`.

Cambia la línea 18 para usar el método `next()` en lugar de `nextLine()`. ¿Qué hace ahora el programa?

Leyendo archivos de texto

Un `BufferedReader` se parece a un `Scanner`, pero se especializa en archivos. Hay cosas que puedes hacer en un archivo, que no podrías hacer en terminal, como regresar a la línea anterior o leer todo el archivo en una sola pasada. El demo siguiente lee un archivo de texto llamado `texto.txt` y lo imprime línea por línea. Asegúrate de crear uno al lado de este código, para que puedas probarlo. Observa que si olvidas hacerlo, este método lanza una excepción, para avisar que el archivo no está y el método `main` la cacha, para convertirla en un mensaje más amigable con el usuario.

```
1  import java.io.BufferedReader;
2  import java.io.FileNotFoundException;
3  import java.io.FileReader;
4  import java.io.IOException;
5
6  /**
7   * Demo básico sobre el uso de FileReader y BufferedReader.
8   * @author blackzafiro
9   */
10 public class DemoBufferedReader {
11   /**
12     * Lee un archivo de texto e imprime su contenido línea por lí
       ↪ nea.
13     * @param args
14     */
15   public static void main(String[] args) {
16     try {
17       BufferedReader in = new BufferedReader(new FileReader("
           ↪ texto.txt"));
18       String line;
19       while((line = in.readLine()) != null) {
20         System.out.println(line);
21       }
22       in.close();
23     } catch (FileNotFoundException e) {
24       System.err.println("No se encontró el archivo texto.txt ¿
           ↪ Olvidaste crearlo?");
25     } catch (IOException ioe) {
26       System.err.println("Error al leer el contenido de texto.txt
           ↪ ");
27     }
28   }
29
30 }
```

Escribiendo archivos de texto

Para escribir archivos de texto utilizaremos un `PrintStream`. Esta clase desciende de `OutputStream` y de hecho la has usado frecuentemente ¿Recuerdas a `System.out`? Pues bien, éste es un objeto de tipo `PrintStream`. Así que siéntete como en casa. El ejemplo de abajo indica cómo crear uno que dirija la salida a un archivo, en lugar de a la terminal. Observa que la sintaxis del `try` es diferente: la creación del objeto que puede lanzar las excepciones se encuentra entre paréntesis. Esto ayudará también a que el objeto se cierre solo al terminar de ejecutar el bloque de código dentro del `try`. También puedes utilizar esta notación con los otros flujos.

```java
import java.io.FileNotFoundException;
import java.io.PrintStream;

/**
 * Demo básico sobre el uso de PrintStream.
 * @author blackzafiro
 */
public class DemoPrintStream {
  /**
    * Escribe el texto indicado en un archivo.
    * @param args
    */
  public static void main(String[] args) {
    String nombreArchivo = "Salida.txt";
    try (PrintStream fout = new PrintStream(nombreArchivo)) {
      fout.println("Inicio");
      fout.format("Línea %d\n", 1);
      fout.println("Fin");
    } catch (FileNotFoundException fnfe) {
      System.err.println("No se encotró el archivo " +
          ↪ nombreArchivo + " y no pudo ser creado");
    } catch (SecurityException se) {
      System.err.println("No se tiene permiso de escribir en el
          ↪ archivo");
    }
  }

}
```

Actividad 11.3

Copia y ejecuta el código anterior.

Revisa la documentación de `PrintStream`. Observa que si ejecutas el programa otra vez, se borra el contenido del archivo y se sobreescribe. Si quieres abrir un archivo para agregarle cosas, en lugar de borrarlo y escribir de nuevo, necesitarás usar un `FileWriter` o un `FileOutputStream` e indicar en el constructor que

quieres usar la opción `append`.

Ejercicios

Utiliza nuestro conocido `System.out` y la clase `Scanner` para crear una interfaz de texto con tu usuario.

1. Ahora la clase de uso servirá para crear y manejar la interfaz de texto. Crea un objeto `UsoBaseDeDatos` al inicio de tu método `main`.

2. En main, crea un ciclo donde el usuario pueda elegir entre:
   - Crear una base datos.
   - Cargar de disco.
   - Guardar la agenda.
   - Agregar un registro.
   - Buscar por nombre.
   - Buscar por teléfono.
   - Imprimirla toda.

   Puedes agregarle un método a tu clase `UsoBaseDeDatos`, que se encargue de imprimir este menú, y mandarlo llamar cuando sea necesario.

   Prueba que el menú funcione, aunque no haga nada. A continuación crearás métodos auxiliares para que tu objeto tipo `UsoBaseDeDatos` pueda atender las diferentes solicitudes.

3. La clase `UsoBaseDeDatos` deberá tener un método que pida al usuario el nombre, dirección y teléfono para un registro nuevo y devuelva un objeto de tipo `RegistroAgenda`, inicializado con los datos correspondientes. Así mismo, tu clase `BaseDeDatosAgenda` necesitará un método para agregar el registro.

   Cuando el usuario seleccione la opción "agregar un registro", usarás el método nuevo para pedir los datos al usuario y luego el método de la agenda para agregar el registro nuevo.

4. Para buscar un registro, pide el nombre o el teléfono usando `Scanner`. Luego usa los métodos de tu base de datos para encontrar la primer coincidencia. Imprime en pantalla el resultado de la búsqueda.

5. Para guardar la agenda.

   a) Crea en `RegistroAgenda` un método que reciba como parámetro un objeto de tipo `PrintStream` y que con él escriba su nombre, dirección y teléfono en líneas distintas. Imagina que estuvieras imprimiendo el contenido del registro en pantalla pero, en lugar de `System.out` usa el objeto que te pasan como parámetro (no hagas ninguna otra cosa con él).

b) Para guardar toda la agenda vamos a hacer ahora un poco de trampa.
Observa de nuevo el esquema del diseño de la base de datos Figura 10.4 y tu código para buscar por nombre y teléfono. Te darás cuenta de que el código de la función `esEste` se ejecuta sobre cualquier elemento de la lista, mientras que éste no devuelva `true` (dado que al devolver `true` se deja de recorrer el resto de la lista).
Extiende nuevamente a la interfaz `Buscador`, pero ahora tu clase se encargará de guardar los elementos de la lista en un archivo. Llámala `GuardaRegistro`.
   1) El constructor de tu clase debe recibir un `PrintStream` ya abierto como parámetro y lo guardará en un atributo.
   2) El método `esEste` utilzará al `printStream` para escribir con él el contenido del registro. Para ello manda llamar el método para guardar el registro que reciba como parámetro[1] y devuelve `false` siempre.

c) Agrega a la clase `BaseDeDatosAgenda` el método:

```
1 public void guardaAgenda(String nombreArchivo) {
2     ...
3 }
```

En este método crearás el `PrintStream` con la dirección indicada en `nombreArchivo`. Esta cadena deberá contener la dirección absoluta o relativa del archivo donde se guardará la agenda[2]. Al crear el `PrintStream` se abrirá el archivo correspondiente para escritura.
Luego usarás una instancia de la clase `GuardaRegistro` para guardar toda la base de datos, pasando como parámetro a su constructor el `PrintStream` que acabas de crear. Necesitarás del método `encuentra` de `ManejadorDeLista`. El procedimiento será casi idéntico a lo que hacías para buscar registros. Ojo: no debes modificar **nada** que se encuentre en el paquete `util`.

d) La interfaz de usuario requerirá un método correspondiente que solicite al usuario la dirección del archivo donde desea guarda la agenda y mande llamar a `guardaAgenda`.

e) Prueba tu método. Dado que utilizaste objetos que crean archivos de texto, debes poder abrir el archivo creado con cualquier editor de texto y leer los datos que fueron guardados.

6. Agrega un método `carga(String nombreArchivo)` que cree un flujo de entrada (puedes utilizar `BufferedReader` o `Scanner`), que lea los datos de tus archivos como líneas de texto y las utilice para insertar los registros en ese archivo a la agenda. Reutiliza los métodos que ya programaste.

7. Prueba todo tu programa.

[1] Recuerda hacer los castings necesarios.
[2] Recuerda incluir esta explicación en tu documentación.

Preguntas

1. ¿Cómo preferirías que se llamara la interfaz `Buscador` para que su nombre fuera más acorde a todos los usos que le hemos dado? ¿Qué hay del método `encuentra`? Describe qué cambios sería necesario hacerle al código.

12 | Recursividad - Máquina de cambio

Meta

Que el alumno practique el uso de la programación recursiva para resolver un problema sencillo.

Objetivos

Al finalizar la práctica el alumno será capaz de:

1. Plantear un problema en forma recursiva.
2. Identificar el o los casos base en una solución recursiva.
3. Identificar el paso recursivo en una solución recursiva.
4. Programar funciones recursivas en `Java`.

Antecedentes

Una forma natural de almacenar soluciones a problemas planteados en forma recursiva es en forma de listas. En estos casos, cada llamada recursiva deberá agregar o quitar elementos a la lista que reciba como respuesta a una llamada recursiva, según el problema del que se trate. Si el problema recursivo es más complejo, es probable que necesites tener listas de listas.

Desarrollo

Esta práctica consiste en resolver un solo problema en forma recursiva, programar la solución en `Java` y ejecutar algunos ejemplos. El problema es el siguiente:

Digamos que se quiere dar n pesos de cambio a un cliente con monedas de $10, $5, $2 y $1. Calcular todas las formas posibles de dar el cambio.

En este ejercicio necesitas una lista de las posibles formas de dar cambio, donde cada forma está almacenada en un objeto, sin embargo existen otras alternativas. Otra forma de resolver este problema es tener una lista de formas de dar cambio, donde cada forma de dar el cambio es a su vez una lista de monedas. La redacción de los siguientes puntos asume que usarás un objeto para representar la forma de dar cambio.

Ejercicios

1. Diseña el algoritmo para resolver este problema sin programarlo todavía. Escribe la receta.
2. Crea la clase `Cambio`, donde los atributos serán el número de monedas de cada denominación. Ej `numDiez, numCinco, etc..`
3. Crea la clase `MaquinaDeCambio`.
4. Agrega un método que reciba como parámetro el `precio` a pagar y devuelva una lista con todas las formas de dar el cambio. Utiliza la lista de `Object` que programaste para la práctica anterior.
5. Punto extra: si tu solución requería el uso de algún ciclo como `for`, has otra versión que no lo use, sino que sea puramente recursiva. Si tu método original ya era sólo recursivo, tienes este punto automáticamente.

13 | Arreglos y Estadísticas

Los arreglos son regiones contiguas de memoria utilizada para almacenar un número fijo de elementos del mismo tipo. Se caracterizan porque cada elemento está colocado en un posición indizada. En `Java` el primer elemento se encuentra en la posición cero.

Meta

Que el alumno cree arreglos, almacene y opere con datos en ellos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Crear arreglos de un tipo dado.
- Llenar arreglos con datos obtenidos de un medio persistente (archivos).
- Ejecutar operaciones en las que intervengan todos los elementos en un arreglo.

Antecedentes

Hasta ahora hemos utilizado variables para almacenar un solo dato, como en los ejemplos:

```
1 int i = 100;
2 String h = "Hola";
```

Cuando requerimos almacenar un número variable de datos, lo que hicimos fue crear objetos con atributos que conocían la ubicación del elemento siguiente. Para acceder a un elemento dado era necesario recorrer a todos los elementos, desde el primero, hasta encontrar al que buscábamos:

```java
public class Lista {
  private Object dato;
  private Lista siguiente;
  // getters y setters ...
}

public interface Condición {
  boolean satisface(Object o);
}

public class ManejadorDeLista {
  private Lista lista;

  public Object satisface(Condición c) {
    Lista temp = lista;
    while(lista != null) {
      if (c.satisface(lista.getDato()) {
        return lista.getDato()
      }
      temp = lista.getSiguiente();
    }
  }
}
```

Ahora estudiaremos una tercera técnica para almacenar varios datos del mismo tipo cuando sabemos de antemano, con suficiente precisión, cuántos datos queremos guardar. Utilizaremos una sóla variable y el operador `[]` para acceder directamente a cualquiera de los n datos almacenados. Esta técnica son los arreglos.

Arreglos de una dimensión

Es posible solicitar a la computadora que nos conceda una región de memoria contigua, donde almacenaremos datos del mismo tipo. Esto permitirá que, dada la posición en memoria del primer elemento pos_0, se pueda utilizar una fórmula matemática para calcular la ubicación del iésimo elemento:

$$\mathrm{pos}_i = \mathrm{pos}_0 + i \ \mathrm{tamaño}(\mathrm{tipo}) \tag{13.1}$$

`Java` nos devolverá un objeto para manejar este tipo de estructura. La forma de declarar una variable para este objeto es: indicar el tipo de datos que se almacenarán en el arreglo, el símbolo `[]` para indicar que se trata de un arreglo y el nombre de la variable.

```java
int[] números;
```

Se crean utilizando el operador `new`, el tipo de sus elementos y el número de sus elementos:

```
1 números = new int[10];
```

Para acceder al iésimo elemento, se utiliza el operador `[]` y la posición del elemento en el arreglo. Para almacenar un dato o acceder a él, el código se ve así:

```
1 números[3] = 15;
2 System.out.println(números[3]);
```

Después de haber creado el arreglo podemos preguntarle a `Java` de qué tamaño lo creamos:

```
1 int tam = números.length;
2 System.out.println(tam);
```

Si el arreglo es de un tipo primitivo, los datos son almacenados directamente en el arreglo. Si su tipo es una clase, debemos pensar que se almacenan referencias a los objetos en el montículo.

ArrayIndexOutOfBoundsException

Ojo, cuando utilizes arreglos y trates de acceder a los elementos guardados en ellos, debes asegurarte de solicitar un índice válido. Si solicitas un índice negativo o una posición mayor o igual al tamaño del arreglo, `Java` te lanzará una excepción tipo `ArrayIndexOutOfBoundsException`. Esta excepción es de tipo `RunTimeException`, por lo que es tu responsabilidad como programador que nunca llegue a manos del usuario final.

DESARROLLO

Crearás un programa que lea datos numéricos de un archivo de texto y extraiga dos medidas estadísticas de ellos: la *media* y la *varianza*. La media el promedio de los datos:

$$\mu = \frac{1}{N} \sum_{i=0}^{N} a_i \tag{13.2}$$

La varianza caracteriza la dispersión de los datos y requiere a la media para su cálculo:

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N}(a_i - \mu)^2 \tag{13.3}$$

Ejercicios

1. Con el editor de archivos de texto plano de tu preferencia, crea un archivo con las características siguientes:

   a) Al inicio debe haber un número entero con el número de datos que piensas introducir.

   b) Luego un número con decimales por cada renglón, hasta completar el número de datos indicados en el primer renglón.

   El archivo se puede ver como sigue:

Listing 13.1: datos.txt

```
5
1.23
45.89
-3
-23.0
0
```

2. Crea un clase llamada `Estadisticas` cuyo atributo sea una referencia a un arreglo de tipo `double`, con los métodos siguientes:

   - Un método `cargaArreglo` que reciba como parámetro el nombre de un archivo. Debe leer el archivo, utilizando el primer entero crear un arreglo tipo `double` de ese tamaño e insertar los números siguientes en el arreglo. Éste arreglo será asignado al atributo creado anteriormente.
   - Un método `media` que devuelva el promedio de los datos en el arreglo. Ojo, el arreglo podría ser `null` si aún no han llamado el método anterior ¿cómo debes resolver ese caso? Documenta tu solución.
   - Un método `varianza` que devuelva la varianza de los datos en el arreglo. Misma advertencia que en el punto anterior.

3. Agrega una clase de uso `UsoEstadisticas` con un método `main` que reciba como parámetro el nombre del archivo que debe leer. (Este nombre estará almacenado en la posición cero el arreglo `args`). Por ejemplo, el programa se ejecutará con:

```
$ java icc.estadisticas.UsoEstadisticas datos.txt
Media:          4.2239
Varianza:       511.2388
```

14 | Arreglos y Matrices

Un mecanismo natural para representar matrices matemáticas en la computadora es con arreglos de dos dimensiones. En esta práctica harás uso de esa técnica.

Meta

Que el alumno aprenda a utilizar arreglos como estructura para almacenar, recuperar y manipular información de objetos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Crear arreglos de dos dimensiones.
- Recorrer los arreglos, visitando elemento por elemento, para creación y manipulación de datos.
- Identificar el tamaño de los arreglos en todas sus dimensiones.
- Poner en práctica sus conocimientos sobre el uso de ciclos para realizar operaciones con arreglos.
- Utilizar arreglos como atributos de objetos.

Antecedentes

Arreglos 2D

Para representar arreglos de más dimensiones, `Java` utiliza arreglos de arreglos. Es decir, arreglos de referencias a arreglos con una dimensión menos. Así los arreglos 2D son arreglos de arreglos 1D. Para crearlos sólo hay que agregar más corchetes y

especificar más tamañanos. Para los arreglos 2D especificaremos primero el número de renglones y luego el número de columnas.

```
// Arreglo con cinco renglones y tres columnas.
double números = new double[5][3];
```

Accederemos a cada posición indicando el renglón y la columna que nos interesan.

```
// Accede al dato en el renglón cero, columna 1.
double unDoble = números[0][1];
```

Arreglos como atributos

Podemos utilizar variables tipo arreglo del mismo modo que cualquier otra variable.

Listing 14.1: CajitaRectangular.java

```
public class CajitaRectangular {
  int números[][];

  public Cajita(int renglones, int columnas) {
    números = new int[renglones][columnas];
  }

  public int getRenglones() {
    return números.length;
  }

  // Cuidado con este.
  public int getColumnas() {
    if (números.length > 0) {
      return números[0].length
    } else {
      return 0;
    }
  }
}
```

Ejercicios

1. Crea una clase llamada `Matriz2D`. Como atributo tendrá un arreglo 2D de tipo `double`. En el constructor se deberá indicar el número de renglones y columnas que tendrá la matriz.

2. Sobreescribe el método `toString` para que, al llamar `System.out.println(`*`matriz`*`)` se imprima en pantalla el contenido de la matriz.

3. Agrega dos métodos: uno que devuelva el número de renglones y otro, el número de columnas.

4. Crea un método estático que reciba un arreglo como parámetro con los datos que se le asignarán a la matriz y devuelva una `Matriz2D` con esos datos. Ojo: debes revisar que tenga las dimensiones correctas para cada renglón.

5. Prueba tu código creando e imprimiendo algunas matrices.

6. Implementa las siguientes operaciones de matrices (recuerda probar cada uno de ellos en una clase de uso):

   a) Suma de matrices: recibe como parámetro otro objeto tipo `Matriz2D` y devuelve una matriz nueva con el resultado.

   TIP: Los atributos privados de otros objetos de la misma clase son accesibles directamente con el operador `..`

   b) Resta de matrices: recibe como parámetro otro objeto tipo `Matriz2D` y devuelve una matriz nueva con el resultado.

   c) Multiplicar una matriz por un escalar: recibe como parámetro un `double` y devuelve una matriz nueva con el resultado.

   d) Multiplicación de matrices: recibe como parámetro otra `Matriz2D` y devuelve una matriz nueva con el resultado.

15 | Interfaces gráficas

16 | Hilos de ejecución y enchufes

Parte II
Proyectos

17 | Sistema solar

Prerrequisitos

- Herencia

Meta

Que el alumno desarrolle una aplicación completa aplicando el diseño orientado a objetos para resolver un problema donde cooperan varias instancias de diversas clases.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Describir el diseño de un proyecto completo utilizando UML.
- Explicar cómo se distribuyen tareas entre diferentes instancias(objetos) de diversas clases.
- Crear una animación en `JavaFX`.

Antecedentes

`JavaFX` es un conjunto de paquetes gráficos y multimedia (video, audio, internet e interfaces gráficas de usuario) para diseñar, crear, probar, purgar y desplegar aplicaciones que operan en cualquier sistema operativo. Su diseño está inspirado en técnicas estándar para la produción de animaciones.

Para quienes tienen `Java` 11.

Actividad 17.1

A partir de Java 11, la reglas cambian. Ahora la recomendación sí es quedarse con OpenJDK. En Ubuntu 18 usa:

```
$ sudo apt install default-jdk
```

Ahora JavaFX ya no es parte de la distribución, así que es necesario instalarlo por separado. Sigue las instrucciones en https://openjfx.io/openjfx-docs/#install-javafx. Para compilar y ejecutar el código ya no funciona ant, lo más fácil por el momento será hacerlo a mano:

```
$ export PATH_TO_FX=<where_you_placed_it>/javafx-sdk-11.0.1/
    ↪ lib
$ javac --module-path $PATH_TO_FX --add-modules=javafx.
    ↪ controls -d ./classes --source-path ./src src/
    ↪ sistemasolar/SistemaSolar.java
$ java --module-path $PATH_TO_FX --add-modules=javafx.controls
    ↪  -classpath classes sistemasolar.SistemaSolar
```

Arquitectura del proyecto

Para programar una aplicación de JavaFX se debe comenzar con tres elementos:

Application La clase principal debe extender javafx.application.Application. La clase base se encargará de toda la configuración previa al lanzamiento de la interfaz de usuario, incluyendo todas las llamadas al sistema operativo, que se esconden detrás. El código correspondiente a tu programa comenzará a ejecutarse a partir de método public void start(Stage primaryStage) throws Exception, que debes sobreescribir.

Stage El método start de la clase Application recibe como parámetro un *escenario*. Esta clase representa al lugar donde se montará tu *escena*. Una aplicación puede tener varias escenas y éstas se van representando en el escenario, puedes alternar entre escenas, en forma análoga a lo que sucede en un teatro o una película.

Scene La *escena* es el contenedor para todos los objetos que se mostrarán al usuario, con sus animaciones.

Nodos

En JavaFX todos los elementos que aparecen en la pantalla se agregan a una escena y son objetos de tipo Node, incluso la cámara que *filma* la escena, aunque no la veamos. La Figura 17.1 muestra los elementos más importantes de la jerarquía de clases.

La documentación oficial se encuentra en https://openjfx.io/javadoc/11/, la necesitarás para consultar los detalles sobre los objetos y métodos de este proyecto.

Ejercicios

Para este proyecto, sólo habrá una escena: el sistema solar con 9 planetas, los satélites más importantes y algunos asteroides (opcionalmente).

1. Para este proyecto se te entrega un aplicación de JavaFX que funciona. Para correrlo en tu computadora descomprime el archivo y abre el build.xml con tu editor de texto preferido. Busca la línea que dice:

```
1   <property name="JAVA_HOME" value="/home/blackzafiro/
      ↪ Descargas/Aplicaciones/Java/jdk-10.0.2"/>
```

   debes sustituir el valor de value por la ruta donde tienes instalado Java. Si todo funciona bien, al ejecutar ant deberás ver una animación con el sol, la Tierra, la Luna y un curioso elemento extra: un satélite para la Luna.

   Tu trabajo será completar el sistema solar, para ello debes leer todo el código que te fue proporcionado y entender cómo funciona. La Figura 17.2 muestra el diagrama de clases, para que veas cómo está organizado el código.

2. Quita al satélite extra, la Luna no lo tiene.

3. Agrega a los planetas faltantes desde Mercurio hasta Neptuno. Los planetas actuales están a escala del sistema real aproximadamente, pero puede suceder que al agregar más planetas ya no quepan todos y no se vea bien. Es válido modificar los valores en aras de mejorar la estética, aunque la animación no sea astronómicamente correcta.

4. Ponle colores a los planetas, según sus colores reales. Se dará un punto extra a quien le ponga texturas con fotos de los planetas.

5. Agrega algunos satélites importantes, como los cuatro de Júpiter que vió Galileo con su telescopio.

6. Guarda una copia de tu sistema ya funcionando antes de proceder al siguiente paso, en caso de que algo salga mal.

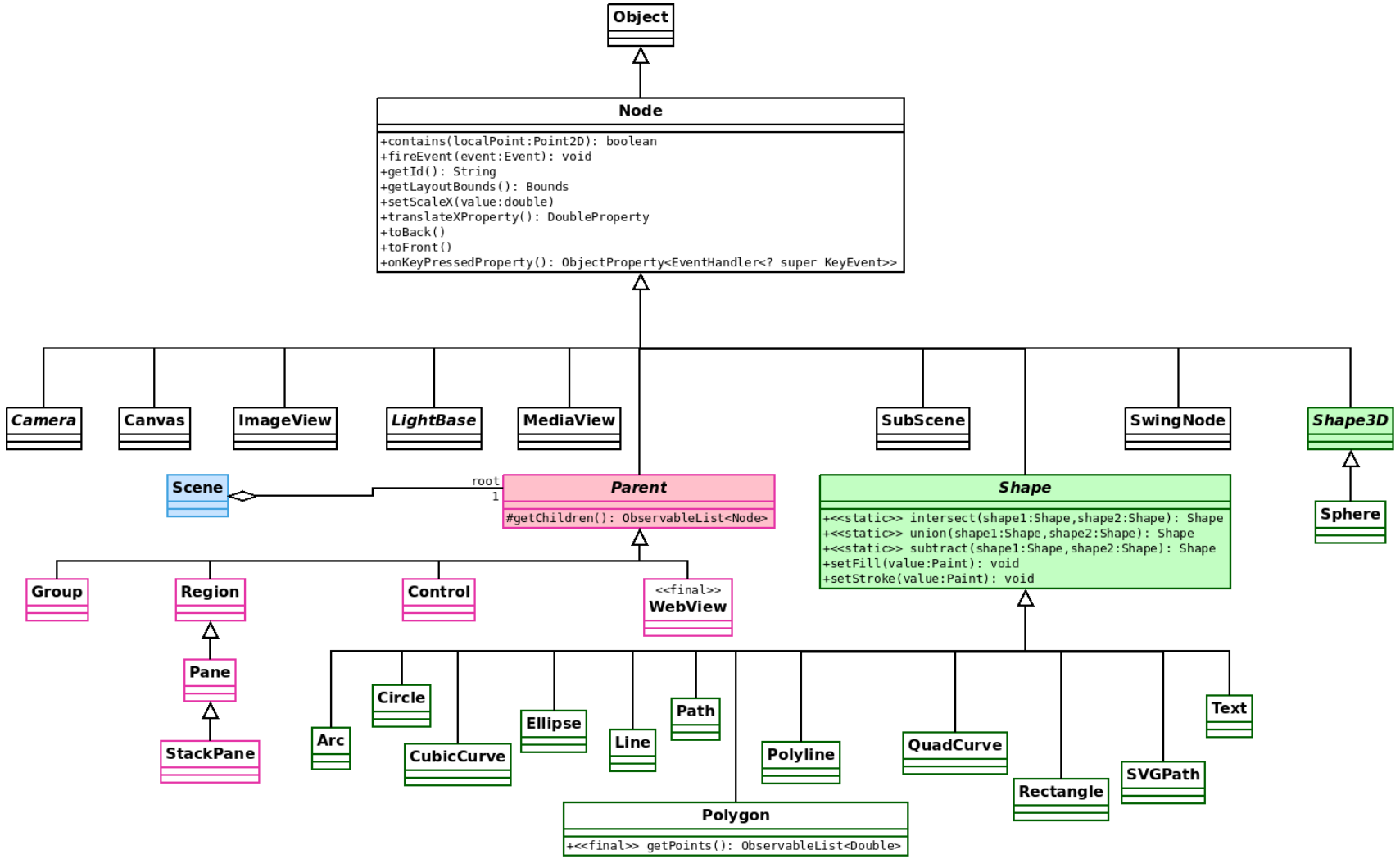

Figura 17.1 Diagrama de clases UML para los nodos de `JavaFX`.

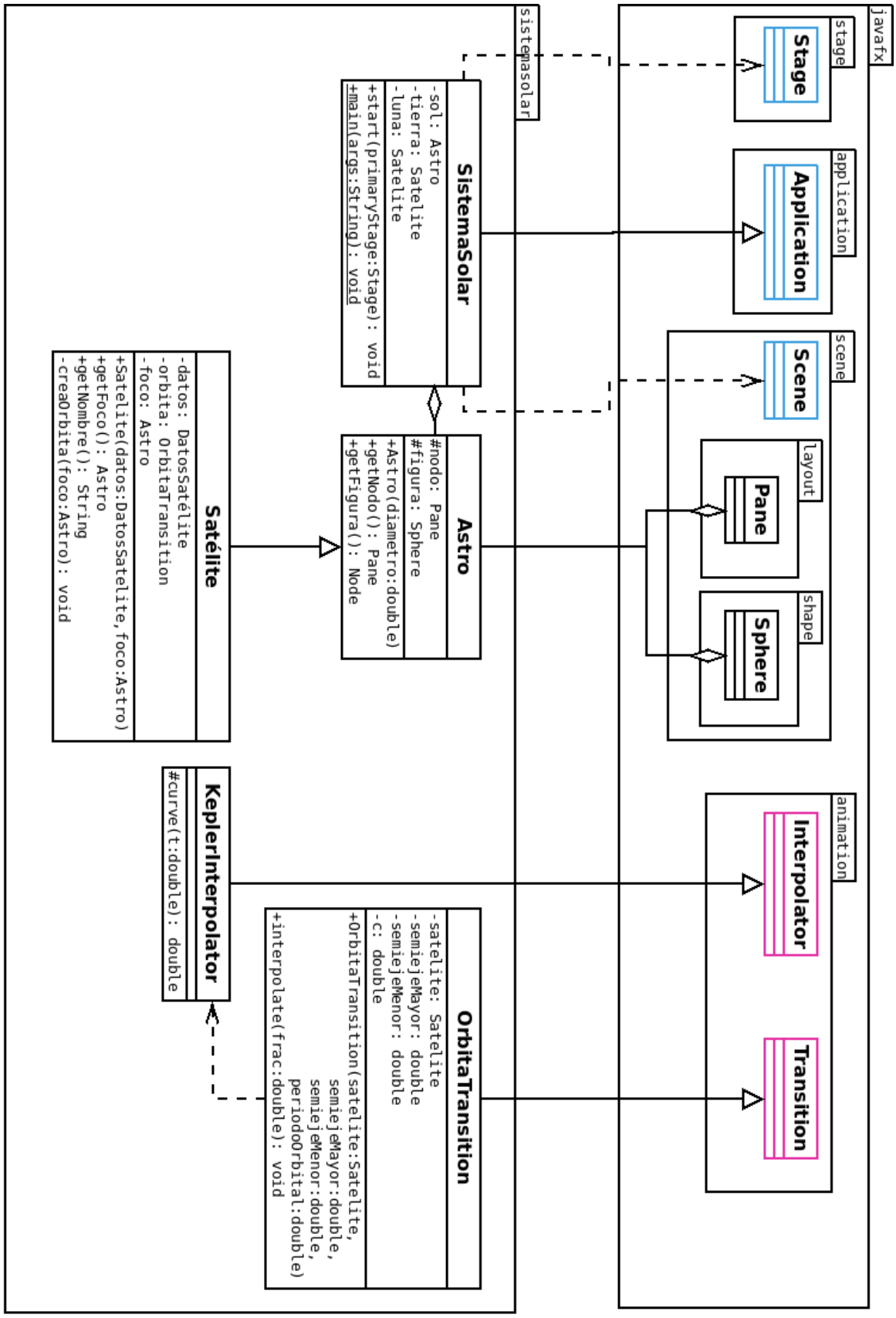

Figura 17.2 Arquitectura de la aplicación de `JavaFX` para modelar al sistema solar.

7. Agrega a Plutón. El aspecto interesante de este planeta enano es que su órbita está inclinada con respecto a las demás, así que necesitarás entender el código que se te proporcionó, para modificarlo y agregar una opción que permita rotar la órbita. Esta parte de la práctica es obligatoria.

8. Se dará otro punto extra a quien genere un número interesante de asteroides entre Marte y Júpiter. Deberás usar ciclos para ello y tal vez algún generador de números aleatorios para que los asteroides se vean un poco dispersos sobre la órbita.

18 | Proyecto: Autómatas celulares bidimensionales

Autor original: José Galaviz Casas
El texto se ha pasado en limpio con adaptaciones.

Prerrequisitos

- Arreglos
- Listas y genéricos
- Herencia

Meta

Que el alumno desarrolle una aplicación completa aplicada a un tema real haciendo uso de orientación a objetos y arreglos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Crear una animación en `JavaFX` gobernada por una simulación matemática.

Antecedentes

En términos generales un sistema dinámico es un modelo que describe el comportamiento, a lo largo del tiempo, de un sistema en función de sus estados previos. Un caso

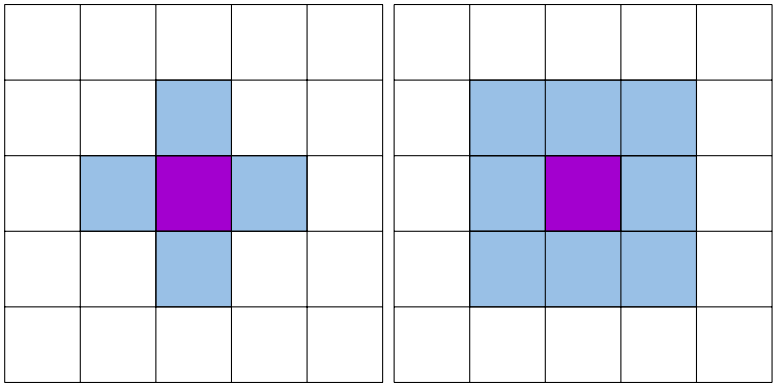

Figura 18.1 Izquierda: vecindad de Von Neumann. Derecha: vecindad de Moore. Los dos tipos de vecindades más usuales en autómatas celulares bidimensionales. El estado de la celda central es determinado, en el siguiente paso temporal, por su valor actual y el de las celdas coloreadas a su alrededor.

particular de un sistema dinámico son los sistemas dinámicos discretos, en lo que el tiempo no es una variable continua sino que avanza a "pasos". En general un sistema dinámico discreto está descrito completamente por la función $f : \mathbb{X} \to \mathbb{X}$ que permite determinar el estado actual del sistema x_t en función de su estado previo x_{t-1}, es decir $x_t = f(x_{t-1})$, donde x_{t-1} y x_t son dos estados de un conjunto de posibles estados del sistema $\mathcal{X}$, en notación: $x_{t-1}, x_t \in \mathcal{X}$.

Un autómata celular es un sistema dinámico discreto, en el que de hecho tanto el espacio como el tiempo y los posibles estados del sistema son discretos. El espacio se divide en *celdas* regulares de igual tamaño y forma, lo que constituye una *malla*. En cualquier instante del tiempo t cada celda de la malla posee un valor determinado de un conjunto de posibles valores, lo que constituye un *estado*. Para determinar el estado de la celda en el siguiente paso temporal (en $t + 1$), se aplica una *regla local*, una función que determina, con base en el valor de la celda misma y de algunas de sus celdas vecinas al tiempo t, el nuevo valor del estado de la celda.

Por supuesto, dado que el estado de una celda está en función de su estado previo y del estado previo de sus celdas vecinas conviene especificar quienes son sus celdas vecinas. Hay una infinidad de opciones, pero las más usuales en los autómatas celulares bidimensionales son dos: la vecindad de Von Neumann y la de Moore. La vecindad de Von Neumann de una celda está constituida por aquellas celdas que se encuentran arriba, abajo, a la derecha y a la izquierda de la celda en cuestión. La de Moore considera además las que están arriba a la derecha, arriba a la izquierda, abajo a la derecha y abajo a la izquierda. En la Figura 18.1 se ilustran los dos tipos de vecindad.

Los autómatas celulares han recibido mucha atención porque han resultado ser modelos muy adecuados para describir el comportamiento de algunos fenómenos naturales muy complejos, con unas cuantas reglas locales muy simples. Desde el punto de vista teórico resultan ser también muy interesantes dado que son un ejemplo de

sistemas con interacciones no lineales: el conocer cómo operan locamente no permite muchas veces conocer el commportamiento general del autómata a largo plazo: reglas simples que engendran comportamientos complejos, algo que suele calificarse como *propiedades emergentes*. Los autómatas celulares resultan estar muchas veces emparentados con los procesos caóticos o en la frontera del caos, con fenómenos de criticalidad auto-organizada y en general con lo que se denominan sistemas complejos.

JavaFX

En este proyecto tendrás que realizar el programa completo. Para que tengas una idea de cómo puedes empezar, se te entrega un pequeño demo.

Actividad 18.1

Si tienes `Java` 10, puedes usar `ant` para ejecutarlo. Si tienes `Java` 11, usa los comandos siguientes:

```
$ export PATH_TO_FX=<where_you_placed_it>/javafx-sdk-11.0.1/
    ↪ lib
$ javac --module-path $PATH_TO_FX --add-modules=javafx.
    ↪ controls -d ./classes --source-path ./src src/automatas/
    ↪ Demo.java
$ java --module-path $PATH_TO_FX --add-modules=javafx.controls
    ↪  -classpath classes automatas.Demo
```

Ojo, `ant clean` aún te servirá para entregar tu código sin archivos compilados.

Ejercicios

Hay varios modelos basados en autómatas celulares que podemos programar, todos requieren la elaboración previa de las mismas clases.

Opción 1: Actividad sísmica

Se utilizan vecindades de Von Neumann. Cada celda puede tomar valores en el conjunto $\mathbb{X} = \{0, ..., M\}$. A M se le llama *valor umbral*.

1. En $t = 0$ la malla se inicializa con valores aleatorios de entre los valores válidos inferiores a M.

2. En cada paso temporal t se elige aleatoriamente un sitio de la malla y su valor se incrementa en una unidad.

   Otra opción es incrementar los valores de todas las celdas, esta última opción evoluciona más rápido.

3. Si alguna celda alcanzó el valor umbral en $t-1$, para t decrementa su valor en cuatro unidades.

4. Si un sitio tiene vecinos en valor umbral en $t-1$, para t incrementa su propio valor en una unidad por cada vecino en valor umbral (sin rebasar el valor umbral).

Los sitios (celdas) de la malla con valor umbral modelan aquellos lugares de una falla tectónica que han acumulado mucha energía y ya no pueden más. Cuando un sitio de la falla alcanza el umbral lo único que le queda por hacer es romperse liberando energía que deben absorber, si pueden, sus sitios vecinos. Cuando muchos sitios se rompen al mismo tiempo se genera un sismo, la magnitud de éste está en función del número de celdas que se rompen.

Opción 2: Un modelo de propagación de epidemias

Se utilizan también vecindades de Von Neumann. Los valores de las celdas están en el conjunto $\mathbb{X} = \{0, ..., a+g\}$, dividido en tres subconjuntos: $A = \{0\}$, $B = \{1, ..., a\}$ y $C = \{a+1, ..., a+g\}$, el conjunto A es *susceptible*, el B es *infeccioso* y el C es *inmune*.

1. Un individuo susceptible se vuelve infeccioso si al menos un vecino es infeccioso.
2. Un individuo inmune después de g pasos se vuelve susceptible.
3. Un individuo que ha sido infeccioso o inmune por menos de g pasos sólo incrementa su valor.
4. Un individuo infeccioso después de a pasos se vuelve inmune.
5. Un individuo susceptible sin vecinos infecciosos permanece susceptible.

Opción 3: Un modelo de incendios forestales

El modelo que se describe a continuación se suele llamar incendio forestal estocástico.

Los valores de las celdas son 0 (*vacío*), 1 (*árbol*) y 2 (*árbol incendiado*), se utilizan vecindades de Von Neumann. Este modelo es el más complicado porque es estocástico,

esto es, se utilizan tres diferentes parámetros probabilísticos que controlan el proceso: p probabilidad de que cresta un árbol $(0 \to 1)$, f probabilidad de que un árbol se incendie espontáneamente $(1 \to 2)$ y g probabilidad de que un árbol sea inmune al fuego. Las reglas son las siguientes:

1. Un sitio 0 se vuelve 1 con probabilidad p.
2. Un sitio 1 se vuelve 2 con probabilidad $(1 - g)$ si al menos un vecino está en estado 2.
3. Un sitio 1 se vuelve 2 con probabilidad $f * (1 - g)$ si no hay vecinos en estado 2.
4. Un sitio 2 se vuelve 0 al siguiente paso temporal.

Generalidades

El proyecto debe implementar, al menos, dos de los tres modelos planteados aquí, se dará máximo un punto extra a quien implemente correctamente los tres.

1. Hacer una clase abstracta de autómatas celulares que implemente las cosas que hay que hacer en todos los casos y que permita fijar la firma de algunos métodos; por ejemplo, uno que actualice el estado de una celda dado su estado actual y el de sus vecinos.
2. Debe presentarse gráficamente la evolución temporal del autómata. Las celdas en diferente estado deben distinguirse por su color. Los colores se dejan a elección del programador.
3. Cuando la simulación termine debe también mostrar una gráfica del número de celdas en estado crítico contra el número de pasos temporales (i.e. abscisas=paso temporal, ordenadas=número de celdas en estado crítico en ese paso temporal). Los estados criticos son:
   - Para el modelo de sismos: las celdas con energía en umbral.
   - Para el modelo de epidemias: las celdas infecciosas.
   - Para el modelo de incendio forestal: las celdas incendiadas.
4. Implementar un menú (puede ser en la consola), que pida al usuario qué modelo desea correr y con qué parámetros, antes de lanzar la simulación.

Entrada

- El número de pasos temporales que el usuario desea dejar evolucionar el sistema.

- El tamaño de la malla.
- Parámetros de control para el autómata (umbrales, valores de cambio de estado, etc.)

Salida

- El desplegado gráfico de la evolución del autómata.
- El desplegado de la gráfica de puntos críticos vs. tiempo.

REFERENCIAS

Bak, P. y K. Chen (1991). «Self-Organized Criticality». En: *Scientific American* 264. Una de las fuentes primigenias del tema de la criticalidad auto-organizada. En la hemeroteca de la Facultad está la colección de *Scientific American* desde hace unos 50 años a la fecha (nota del 2003)., págs. 46-53.

Gaylord, Richar J. y Kazume Nishidate (1996). «Cellular Automata Simulations with Mathematica». En: *Modelling Nature*. La colocación en la biblioteca es QA267.5,C45,G39. Incluye muchos ejemplos más de autómatas celulares como modelos de fenómenos naturales. TELOS-Springer Verlag.

Nakanishi, Hiizu (jun. de 1990). «Cellular-automaton model of earthquakes with deterministic dynamics». En: *Physical Review A* 41.12. Una generalización del modelo presentado aquí. Además se muestra que los resultados arrojados por éste son consistentes con la ley de Gutenberg-Richter, el modelo continuo más robusto en terremotos., págs. 7086-7089.

Schönfisch, Birgitt (1995). «Propagation of Fronts in Cellular Automata». En: *Physica D: Nonlinear Phenomena* 80. Incluye un par de modelos de propagación de epidemias., págs. 433-450.

Bibliografía

Aho, Alfred V., Ravi Sethi y Jeffrey D. Ullman (1998). *Compiladores. Principios, técnicas y herramientas.* Trad. por Pedro Flores Suárez y Pere Botella i López. Pearson, Addison Wesley Longman.

Fish (sep. de 2005). *Float*. Inglés. Apple. URL: http://ridiculousfish.com/blog/posts/float.html.

I/O Streams (2016). URL: https://docs.oracle.com/javase/tutorial/essential/io/streams.html.

López Gaona, Amparo (2012). *Introducción al desarrollo de programas con Java. Prácticas.* Las prensas de Ciencias.

Viso, Elisa y Canek Peláez (2007). *Introducción a las Ciencias de la Computación con Java*. Las prensas de ciencias.

– (jun. de 2012). *Introducción a las Ciencias de la Computación con Java*. 2a. Temas de computación. Las prensas de ciencias.