Java™ CÓMO PROGRAMAR

Décima edición

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Traducción
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Instituto Tecnológico y de Estudios Superiores de Monterrey - Campus Monterrey

Revisión técnica
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Judith Sonck Ledezma
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Departamento de Sistemas
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, Instituto Politécnico Nacional, México

Pearson

15

Archivos, flujos y serialización de objetos

La conciencia … no aparece a sí misma cortada en pequeños pedazos. … Un "río" o un "flujo" son las metáforas por las cuales se describe con más naturalidad.

—William James

Objetivos

En este capítulo aprenderá:

- A crear, leer, escribir y actualizar archivos.
- A obtener información de los archivos y directorios mediante las funciones de las API NIO.2.
- A conocer las diferencias entre los archivos de texto y los archivos binarios.
- A utilizar la clase `Formatter` para enviar texto a un archivo.
- A utilizar la clase `Scanner` para recibir texto de un archivo.
- A escribir y leer objetos de un archivo mediante el uso de la serialización de objetos y la interfaz `Serializable`, además de las clases `ObjectInputStream` y `ObjectOutputStream`.
- A utilizar un cuadro de diálogo `JFileChooser` para que los usuarios puedan seleccionar archivos o directorios en un disco.

Plan general

15.1 Introducción

El almacenamiento de datos en variables y arreglos es *temporal*, ya que los datos se pierden cuando una variable local queda fuera de alcance o cuando el programa termina. Las computadoras utilizan **archivos** para retener datos a largo plazo, incluso después de que los programas que crean esos datos terminan. Usted utiliza archivos a diario para tareas como escribir un documento o crear una hoja de cálculo. Las computadoras almacenan archivos en **dispositivos de almacenamiento secundario** como discos duros, unidades flash, DVD y más. Los datos que se mantienen en archivos son **datos persistentes** ya que permanecen más allá de la ejecución del programa. En este capítulo explicaremos cómo los programas en Java crean, actualizan y procesan archivos.

Empezaremos con un análisis sobre la arquitectura de Java para manejar archivos mediante programación. Luego explicaremos que los datos pueden almacenarse en *archivos de texto* y *archivos binarios*, y cubriremos las diferencias entre ellos. Demostraremos cómo obtener información sobre archivos y directorios mediante el uso de las clases `Paths` y `Files`, y mediante las interfaces `Path` y `DirectoryStream` (todas del paquete `java.nio.file`). Después consideraremos los mecanismos para escribir y leer datos en archivos. Le mostraremos cómo crear y manipular archivos de texto de acceso secuencial. Al trabajar con archivos de texto, el lector puede empezar a manipular archivos fácil y rápidamente. Sin embargo, como veremos más adelante es difícil leer datos de los archivos de texto y devolverlos al formato de los objetos. Por fortuna muchos lenguajes orientados a objetos (incluyendo Java) ofrecen distintas formas de escribir objetos en (y leer objetos de) archivos, lo cual se conoce como *serialización* y *deserialización de objetos*. Para demostrar esto recreamos algunos de nuestros programas de acceso secuencial que utilizaban archivos de texto pero esta vez almacenando y recuperando objetos de archivos binarios.

15.2 Archivos y flujos

Java considera a cada archivo como un **flujo de bytes** secuencial (figura 15.1).[1] Cada sistema operativo proporciona un mecanismo para determinar el fin de un archivo, como el **marcador de fin de archivo** o la cuenta de bytes totales en el archivo que se registra en una estructura de datos administrativa, mantenida por el sistema. Un programa de Java que procesa un flujo de bytes simplemente recibe una indicación del sistema operativo cuando el programa llega al fin del flujo; el programa *no* necesita saber cómo representa la plataforma subyacente a los archivos o flujos. En algunos casos, la indicación de fin de

1. Las API NIO de Java también incluyen clases e interfaces que implementan lo que se conoce como la arquitectura basada en canales para las operaciones de E/S de alto rendimiento. Estos temas están más allá del alcance de este libro.

archivo ocurre como una excepción. En otros casos, la indicación es un valor de retorno de un método invocado en un objeto procesador de flujos.

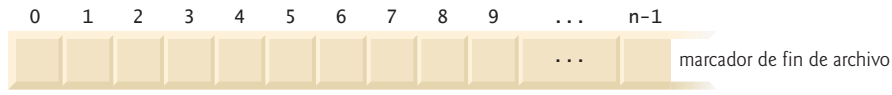

Fig. 15.1 | La manera en que Java ve a un archivo de *n* bytes.

Flujos basados en bytes y basados en caracteres

Los flujos de archivos se pueden utilizar para la entrada y salida de datos, ya sea como bytes o como caracteres.

- Los **flujos basados en bytes** reciben y envían datos en su formato *binario*; un `char` es de dos bytes, un `int` es de cuatro bytes, un `double` es de ocho bytes, etcétera.
- Los **flujos basados en caracteres** reciben y envían datos como una *secuencia de caracteres* en la que cada carácter es de dos bytes; el número de bytes para un valor dado depende del número de caracteres en ese valor. Por ejemplo, el valor `2000000000` requiere 20 bytes (10 caracteres a 2 bytes por carácter), pero el valor `7` requiere sólo dos bytes (1 carácter a dos bytes por carácter).

Los archivos que se crean usando flujos basados en bytes se conocen como **archivos binarios**, mientras que los archivos que se crean usando flujos basados en caracteres se conocen como **archivos de texto**. Los archivos de texto se pueden leer con editores de texto, mientras que los archivos binarios se leen mediante programas que comprenden el contenido específico del archivo y su orden. Un valor numérico en un archivo binario se puede utilizar en cálculos, mientras que en una cadena de texto, por ejemplo, el carácter `5` es simplemente un carácter que puede utilizarse como en "`Sarah Miller tiene 15 años de edad`".

Flujos estándar de entrada, salida y error

Un programa de Java **abre** un archivo creando un objeto y asociándole un flujo de bytes o de caracteres. El constructor del objeto interactúa con el sistema operativo para *abrir* el archivo. Java también puede asociar flujos con distintos dispositivos. Cuando un programa de Java empieza a ejecutarse crea tres objetos flujo que se asocian con dispositivos: `System.in`, `System.out` y `System.err`. Por lo general, `System.in` (el objeto flujo estándar de entrada) permite a un programa recibir bytes desde el teclado. El objeto `System.out` (el objeto flujo estándar de salida) generalmente permite a un programa mostrar datos en la pantalla. El objeto `System.err` (el objeto flujo estándar de error) normalmente permite a un programa mostrar en la pantalla mensajes de error basados en caracteres. Cada uno de estos flujos puede **redirigirse**. Para `System.in`, esta capacidad permite al programa leer bytes desde un origen distinto. Para `System.out` y `System.err` esta capacidad permite que la salida se envíe a una ubicación distinta, como un archivo en disco. La clase `System` proporciona los métodos `setIn`, `setOut` y `setErr` para redirigir los flujos estándar de entrada, salida y de error, respectivamente.

Los paquetes `java.io` y `java.nio`

Los programas de Java realizan el procesamiento de archivos mediante las clases e interfaces del paquete `java.io` y los subpaquetes de `java.nio`, que son las nuevas API de E/S de Java que se introdujeron por primera vez en Java SE 6 y que se han ido mejorando desde entonces. Hay también otros paquetes en las API de Java que contienen clases e interfaces basadas en las de los paquetes `java.io` y `java.nio`.

Las operaciones de entrada y salida basadas en caracteres se pueden llevar a cabo con las clases `Scanner` y `Formatter`, como veremos en la sección 15.4. Hemos usado la clase `Scanner` con mucha frecuencia para recibir datos del teclado, pero esta clase también puede leer datos desde un archivo. La clase `Formatter` permite mostrar datos con formato a cualquier flujo basado en texto, en forma similar al método `System.out.printf`. En el apéndice I se presentan los detalles acerca de la salida con formato mediante `printf`. Todas estas características se pueden utilizar también para dar formato a los archivos de texto. En el capítulo 28 (en inglés, en el sitio web del libro) usaremos las clases de flujos para implementar aplicaciones de redes.

Java SE 8 agrega otro tipo de flujo

En el capítulo 17, Lambdas y flujos de Java SE 8, presentaremos un nuevo tipo de flujo que se utiliza para procesar colecciones de elementos (como arreglos y objetos `ArrayList`), en vez de los flujos de bytes que veremos en los ejemplos de procesamiento de archivos de este capítulo.

15.3 Uso de clases e interfaces NIO para obtener información de archivos y directorios

Las interfaces `Path` y `DirectoryStream`, junto con las clases `Paths` y `Files` (todas del paquete `java.nio.file`) son útiles para recuperar información sobre los archivos y directorios en el disco:

- La interfaz **`Path`**. Los objetos de las clases que implementan esta interfaz representan la ubicación de un archivo o directorio. Los objetos `Path` no abren archivos ni proporcionan herramientas de procesamiento de archivos.
- La clase **`Paths`**. Proporciona métodos `static` que se usan para obtener un objeto `Path`, el cual representa la ubicación de un archivo o directorio.
- La clase **`Files`**. Proporciona métodos `static` para manipulaciones comunes de archivos y directorios, como copiar archivos, crear y eliminar archivos y directorios, obtener información sobre archivos y directorios, leer el contenido de archivos, obtener objetos que permitan manipular el contenido de los archivos y directorios, y más.
- La interfaz **`DirectoryStream`**. Los objetos de las clases que implementan esta interfaz permiten a un programa iterar a través del contenido de un directorio.

Creación de objetos `Path`

Podrá usar el método `static` **`get`** de la clase `Paths` para convertir un objeto `String`, que representa la ubicación de un archivo o directorio, en un objeto `Path`. Después podrá usar los métodos de la interfaz `Path` y la clase `Files` para determinar información sobre el archivo o directorio especificado. A continuación hablaremos sobre varios de esos métodos. Para las listas completas de sus métodos, visite:

```
http://docs.oracle.com/javase/7/docs/api/java/nio/file/Path.html
http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html
```

Comparación entre rutas absolutas y rutas relativas

La ruta de un archivo o directorio especifica su ubicación en el disco. La ruta incluye algunos o todos los directorios que conducen a ese archivo o directorio. Una **ruta absoluta** contiene *todos* los directorios —empezando con el **directorio raíz**— que conducen a un archivo o directorio específico. Cada archivo o directorio en un disco duro específico tiene el *mismo* directorio raíz en su ruta. Una **ruta relativa** es "relativa" al directorio actual; es decir, se trata de una ruta que está en función del directorio en el que la aplicación empezó a ejecutarse.

Obtener objetos Path a partir de objetos URI

Una versión sobrecargada del método `static get` de `Files` usa un objeto `URI` para localizar el archivo o directorio. Un **identificador uniforme de recursos (URI)** es una forma más general de un **localizador uniforme de recursos (URL)**, el cual se utiliza para localizar sitios Web. Por ejemplo, `http://www.deitel.com/` es el URL para el sitio Web de Deitel & Associates. Los URI para localizar archivos varían entre los distintos sistemas operativos. En plataformas Windows, el URI:

```
file://C:/datos.txt
```

identifica al archivo `datos.txt`, almacenado en el directorio raíz de la unidad C:. En plataformas UNIX/Linux, el URI

```
file:/home/estudiantes/datos.txt
```

identifica el archivo `datos.txt` almacenado en el directorio `home` del usuario `estudiante`.

Ejemplo: cómo obtener la información de archivos y directorios

La figura 15.2 pide al usuario que introduzca el nombre de un archivo o directorio, y después usa las clases `Paths`, `Path`, `Files` y `DirectoryStream` para imprimir información en pantalla acerca de ese archivo o directorio. El programa empieza pidiendo al usuario un archivo o directorio (línea 16). En la línea 19 se introduce el nombre del archivo o directorio y se pasa al método `static get` de `Paths`, que convierte el objeto `String` en un objeto `Path`. En la línea 21 se invoca el método `static exists` de `File`, el cual recibe un objeto `Path` y determina si el nombre introducido por el usuario existe (ya sea como archivo o directorio) en el disco. Si el nombre no existe, el control procede a la línea 49 que muestra un mensaje en la pantalla que contiene la representación `String` del objeto `Path` seguida de "`no existe`". En caso contrario, se ejecutan las líneas 24 a 45:

- El método `getFileName` de `Path` (línea 24) obtiene el nombre `String` del archivo o directorio sin información sobre la ubicación.
- El método `static isDirectory` de `Files` (línea 26) recibe un objeto `Path` y devuelve un `boolean` que indica si ese objeto `Path` representa un directorio en el disco.
- El método `isAbsolute` de `Path` (línea 28) devuelve un `boolean` que indica si ese objeto `Path` representa una ruta absoluta a un archivo o directorio.
- El método `static getLastModifiedTime` de `Files` (línea 30) recibe un objeto `Path` y devuelve un objeto `FileTime` (paquete `java.nio.file.atribute`) que indica cuándo fue la última vez que se modificó el archivo. El programa imprime en pantalla la representación `String` predeterminada de `FileTime`.
- El método `static size` de `Files` (línea 31) recibe un objeto `Path` y devuelve un `long` que representa el número de bytes en el archivo o directorio. Para los directorios, el valor devuelto es específico de la plataforma.
- El método `toString` de `Path` (que se invoca de manera implícita en la línea 32) devuelve un objeto `String` que representa el objeto `Path`.
- El método `toAbsolutePath` de `Path` (línea 33) convierte el objeto `Path` con el que se invoca en una ruta absoluta.

Si el objeto `Path` representa un directorio (línea 35), en las líneas 40 y 41 se usa el método `static newDirectoryStream` de `Files` (líneas 40 y 41) para obtener un objeto `DirectoryStream<Path>` que contiene objetos `Path` para el contenido del directorio. En las líneas 43 y 44 se muestra la representación `String` de cada objeto `Path` en `DirectoryStream<Path>`. Cabe mencionar que `DirectoryStream` es un tipo genérico como `ArrayList` (sección 7.16).

El primer resultado de este programa demuestra un objeto `Path` para la carpeta que contiene los ejemplos de este capítulo. El segundo resultado demuestra un objeto `Path` para el archivo de código fuente de este ejemplo. En ambos casos especificamos una ruta absoluta.

```
1  // Fig. 15.2: InfoArchivosYDirectorios.java
2  // Clase File utilizada para obtener información sobre archivos y directorios.
3  import java.io.IOException;
4  import java.nio.file.DirectoryStream;
5  import java.nio.file.Files;
6  import java.nio.file.Path;
7  import java.nio.file.Paths;
8  import java.util.Scanner;
9
10 public class InfoArchivosYDirectorios
11 {
12    public static void main(String[] args) throws IOException
13    {
14       Scanner entrada = new Scanner(System.in);
15
16       System.out.println("Escriba el nombre del archivo o directorio:");
17
18       // crear objeto Path con base en la entrada del usuario
19       Path ruta = Paths.get(entrada.nextLine());
20
21       if (Files.exists(ruta)) // si la ruta existe, mostrar en pantalla
                                 información sobre ella
22       {
23          // mostrar información de archivo (o directorio)
24          System.out.printf("%n%s existe%n", ruta.getFileName());
25          System.out.printf("%s un directorio%n",
26             Files.isDirectory(ruta) ? "Es" : "No es");
27          System.out.printf("%s una ruta absoluta%n",
28             ruta.isAbsolute() ? "Es" : "No es");
29          System.out.printf("Fecha de ultima modificacion: %s%n",
30             Files.getLastModifiedTime(ruta));
31          System.out.printf("Tamanio: %s%n", Files.size(ruta));
32          System.out.printf("Ruta: %s%n", ruta);
33          System.out.printf("Ruta absoluta: %s%n", ruta.toAbsolutePath());
34
35          if (Files.isDirectory(ruta)) // imprime en pantalla el listado del
                                            directorio
36          {
37             System.out.printf("%nContenido del directorio:%n");
38
39             // objeto para iterar a través del contenido de un directorio
40             DirectoryStream<Path> flujoDirectorio =
41                Files.newDirectoryStream(ruta);
42
43             for (Path p : flujoDirectorio)
44                System.out.println(r);
45          }
46       }
47       else // no es archivo o directorio, imprimir en pantalla mensaje de error
48       {
```

Fig. 15.2 | Uso de la clase `File` para obtener información sobre archivos y directorios (parte 1 de 2).

```
            System.out.printf("%s no existe%n", ruta);
         }
      } // fin de main
   } // fin de la clase InfoArchivosYDirectorios
```

```
Escriba el nombre del archivo o directorio:
e:\ejemplos\cap15

cap15 existe
Es un directorio
Es una ruta absoluta
Fecha de ultima modificacion: 2013-11-08T19:50:00.838256Z
Tamanio: 4096
Ruta: c:\ejemplos\cap15
Ruta absoluta: c:\ejemplos\cap15

Contenido del directorio:
C:\ejemplos\cap15\fig15_02
C:\ejemplos\cap15\fig15_12_13
C:\ejemplos\cap15\AppsSerializacion
C:\ejemplos\cap15\AppsArchivosTexto
```

```
Escriba el nombre del archivo o directorio:
c:\ejemplos\cap15\fig15_02\InfoArchivosYDirectorios.java

InfoArchivosYDirectorios.java existe
No es un directorio
Es una ruta absoluta
Fecha de ultima modificacion: 2016-05-03T18:26:32Z
Tamanio: 3151
Ruta: c:\ejemplos_codigo\cap15\fig15_02\InfoArchivosYDirectorios.java
Ruta absoluta: c:\ejemplos_codigo\cap15\fig15_02\InfoArchivosYDirectorios.java
```

Fig. 15.2 | Uso de la clase `File` para obtener información sobre archivos y directorios (parte 2 de 2).

Tip para prevenir errores 15.1

Una vez que se confirma que existe un objeto `Path`, aún es posible que los métodos demostrados en la figura 15.2 lancen excepciones `IOException`. Por ejemplo, el archivo o directorio representado por el objeto `Path` podría eliminarse del sistema después de la llamada al método `exists` de `Files` y antes de que se ejecuten las otras instrucciones en las líneas 24 a 45. Los programas de procesamiento de archivos y directorios para uso industrial requieren de un manejo exhaustivo de las excepciones para recuperarse de dichas posibilidades.

Caracteres separadores

Un **carácter separador** se utiliza para separar directorios y archivos en la ruta. En un equipo Windows, el *carácter separador* es la barra diagonal inversa (\). En un sistema Linux o Mac OS X, el carácter separador es la barra diagonal (/). Java procesa ambos caracteres en forma idéntica en el nombre de una ruta. Por ejemplo, si deseamos utilizar la ruta

```
c:\Archivos de programa\Java\jdk1.6.0_11\demo/jfc
```

que emplea a cada uno de los caracteres separadores, Java de todas formas procesa la ruta en forma apropiada.

Buena práctica de programación 15.1

Al construir objetos `String` que representen la información de una ruta, use `File.separator` para obtener el carácter separador apropiado del equipo local, en vez de utilizar `/` o `\` de manera explícita. Esta constante devuelve un objeto `String` que consiste de un carácter que es el separador apropiado para el sistema.

Error común de programación 15.1

Usar `\` como separador de directorios en vez de `\\` en una literal de cadena es un error lógico. Una sola `\` indica que la `\` y el siguiente carácter representan una secuencia de escape. Para insertar una `\` en una literal de cadena, debe usar `\\`.

15.4 Archivos de texto de acceso secuencial

A continuación crearemos y manipularemos *archivos de acceso secuencial* en los que se guardan los registros en un orden basado en el campo clave de registro. Empezaremos con los *archivos de texto* que permiten al lector crear y editar rápidamente archivos que puedan ser leídos por el ser humano. Hablaremos sobre crear, escribir y leer datos, así como actualizar los archivos de texto de acceso secuencial. También incluiremos un programa de consulta de crédito para obtener datos específicos de un archivo. Los programas en las secciones 15.4.1 a 15.4.3 se encuentran en el directorio `AppsArchivosTexto` del capítulo, con lo cual podrá manipular el mismo archivo de texto que también está almacenado en ese directorio.

15.4.1 Creación de un archivo de texto de acceso secuencial

Java no impone una estructura en un archivo; nociones como las de registros no existen como parte del lenguaje Java. Por lo tanto, el programador debe estructurar los archivos de manera que cumplan con los requerimientos de sus aplicaciones. En el siguiente ejemplo veremos cómo imponer una estructura de registros *con claves* en un archivo.

El programa en esta sección crea un archivo simple de acceso secuencial que podría utilizarse en un sistema de cuentas por cobrar para ayudar a administrar el dinero que deben a una compañía los clientes a crédito. Por cada cliente el programa obtiene un número de cuenta, el nombre del cliente y su saldo (es decir, el monto que el cliente aún debe a la compañía por los bienes y servicios recibidos). Los datos obtenidos para cada cliente constituyen un "registro" para ese cliente. El número de cuenta se utiliza como la *clave de registro* en esta aplicación; los registros del archivo se crearán y mantendrán en orden con base en el número de cuenta. El programa supone que el usuario introduce los registros en el orden del número de cuenta. En un sistema completo de cuentas por cobrar (basado en archivos de acceso secuencial), se proporcionaría una herramienta para *ordenar* datos, de manera que el usuario pudiera introducir los registros en *cualquier* orden. Después, los registros se ordenarían y se escribirían en el archivo.

La clase `CrearArchivoTexto`

La clase `CrearArchivoTexto` (figura 15.3) usa un objeto `Formatter` para mostrar en pantalla objetos `String` con formato, usando las mismas herramientas de formato que el método `System.out.printf`. Un objeto `Formatter` puede enviar datos a varias ubicaciones como podría ser una ventana de símbolo del sistema o un archivo, tal como lo hacemos en este ejemplo. El objeto `Formatter` se instancia en la línea 26, en el método `abrirArchivo` (líneas 22 a 38). El constructor que se utiliza en la línea 26 recibe un objeto `String` como argumento, el cual contiene el nombre del archivo que incluye su ruta. Si no se especifica una ruta, como se da aquí el caso, la JVM asume que el archivo está en el directorio desde el cual se ejecutó el programa. Para los archivos de texto, utilizamos la extensión `.txt`. Si el archivo *no* existe, entonces se *creará*. Si se abre un archivo *existente*, su contenido se **trunca** y todos los datos en el archivo se *descartan*. Si no ocurre una excepción, el archivo se abre para escritura y se puede utilizar el objeto `Formatter` resultante para escribir datos en el archivo.

```
1  // Fig. 15.3: CrearArchivoTexto.java
2  // Escribir datos en un archivo de texto secuencial mediante la clase Formatter.
3  import java.io.FileNotFoundException;
4  import java.lang.SecurityException;
5  import java.util.Formatter;
6  import java.util.FormatterClosedException;
7  import java.util.NoSuchElementException;
8  import java.util.Scanner;
9
10 public class CrearArchivoTexto
11 {
12    private static Formatter salida; // envía texto a un archivo
13
14    public static void main(String[] args)
15    {
16       abrirArchivo();
17       agregarRegistros();
18       cerrarArchivo();
19    }
20
21    // abre el archivo clientes.txt
22    public static void abrirArchivo()
23    {
24       try
25       {
26          salida = new Formatter("clientes.txt"); // abre el archivo
27       }
28       catch (SecurityException securityException)
29       {
30          System.err.println("Permiso de escritura denegado. Terminando.");
31          System.exit(1); // termina el programa
32       }
33       catch (FileNotFoundException fileNotFoundException)
34       {
35          System.err.println("Error al abrir el archivo. Terminando.");
36          System.exit(1); // termina el programa
37       }
38    }
39
40    // agrega registros al archivo
41    public static void agregarRegistros()
42    {
43       Scanner entrada = new Scanner(System.in);
44       System.out.printf("%s%n%s%n? ",
45          "Escriba numero de cuenta, nombre, apellido y saldo.",
46          "Para terminar la entrada, escriba el indicador de fin de archivo.");
47
48       while (entrada.hasNext()) // itera hasta encontrar el indicador de fin de
                                  archivo
49       {
50          try
51          {
```

Fig. 15.3 | Escritura de datos en un archivo de texto secuencial mediante la clase `Formatter` (parte 1 de 2).

```
52              // escribe el nuevo registro en el archivo; asume una entrada válida
53              salida.format("%d %s %s %.2f%n", entrada.nextInt(),
54                 entrada.next(), entrada.next(), entrada.nextDouble());
55           }
56           catch (FormatterClosedException formatterClosedException)
57           {
58              System.err.println("Error al escribir en el archivo. Terminando.");
59              break;
60           }
61           catch (NoSuchElementException elementException)
62           {
63              System.err.println("Entrada invalida. Intente de nuevo.");
64              entrada.nextLine(); // descarta la entrada para que el usuario
                intente de nuevo
65           }
66
67           System.out.print("? ");
68        } // fin de while
69     } // fin del método agregarRegistros
70
71     // cierra el archivo
72     public static void cerrarArchivo()
73     {
74        if (salida != null)
75           salida.close();
76     }
77  } // fin de la clase CrearArchivoTexto
```

```
Escriba numero de cuenta, nombre, apellido y saldo.
Para terminar la entrada, escriba el indicador de fin de archivo.
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

Fig. 15.3 | Escritura de datos en un archivo de texto secuencial mediante la clase `Formatter` (parte 2 de 2).

En las líneas 28 a 32 se maneja la excepción tipo **`SecurityException`**, que ocurre si el usuario no tiene permiso para escribir datos en el archivo. En las líneas 33 a 37 se maneja la excepción tipo **`FileNotFoundException`** que ocurre si el archivo no existe y no se puede crear uno nuevo. Esta excepción también puede ocurrir si hay un error al *abrir* el archivo. En ambos manejadores de excepciones podemos llamar al método `static` **`System.exit`** y pasarle el valor `1`. Este método termina la aplicación. Un argumento de `0` para el método `exit` indica la terminación *exitosa* del programa. Un valor distinto de cero, como el `1` en este ejemplo, por lo general indica que ocurrió un error. Este valor se pasa a la ventana de comandos en la que se ejecutó el programa. El argumento es útil si el programa se ejecuta desde un **archivo de procesamiento por lotes** en los sistemas Windows, o una **secuencia de comandos de shell** en sistemas UNIX/Linux/Mac OS X. Los archivos de procesamiento por lotes y las secuencias de comandos de shell ofrecen una manera conveniente de ejecutar varios programas en secuencia. Cuando termina el primer programa, el siguiente programa empieza su ejecución. Se puede usar el argumento para el método `exit` en un archivo de procesamiento por lotes o en una secuencia de comandos de shell, para determinar si deben ejecutarse otros programas. Para obtener más información sobre los archivos de procesamiento por lotes o las secuencias de comandos de shell, consulte la documentación de su sistema operativo.

El método `agregarRegistros` (líneas 41 a 69) pide al usuario que introduzca los diversos campos para cada registro, o la secuencia de teclas de fin de archivo cuando termine de introducir los datos. La figura 15.4 lista las combinaciones de teclas para introducir el fin de archivo en varios sistemas computacionales.

Sistema operativo	Combinación de teclas
UNIX/Linux/Mac OS X	*<Intro> <Ctrl> d*
Windows	*<Ctrl> z*

Fig. 15.4 | Combinaciones de teclas de fin de archivo.

En las líneas 44 a 46 se piden los datos de entrada al usuario. En la línea 48 se utiliza el método `hasNext` de `Scanner` para determinar si se ha introducido la combinación de teclas de fin de archivo. El ciclo se ejecuta hasta que `hasNext` encuentra el fin de archivo.

En las líneas 53 a 54 se usa un objeto `Scanner` para leer los datos del usuario y luego se envían los datos como un registro usando el objeto `Formatter`. Cada método de entrada de `Scanner` lanza una excepción tipo **`NoSuchElementException`** (que se maneja en las líneas 61 a 65) cuando los datos se encuentran en el formato incorrecto (por ejemplo, un objeto `String` cuando se espera un valor `int`), o cuando no hay más datos que introducir. La información del registro se envía mediante el método **`format`**, que puede efectuar un formato idéntico al del método `System.out.printf`, que se utilizó en muchos de los ejemplos de capítulos anteriores. El método `format` envía un objeto `String` con formato al destino de salida del objeto `Formatter`; es decir, el archivo `clientes.txt`. La cadena de formato "`%d %s %s %.2f%n`" indica que el registro actual se almacenará como un entero (el número de cuenta) seguido de un objeto `String` (el nombre), otro `String` (el apellido) y un valor de punto flotante (el saldo). Cada pieza de información se separa de la siguiente mediante un espacio, y el valor tipo `double` (el saldo) se imprime en pantalla con dos dígitos a la derecha del punto decimal (como lo indica el `.2` en `%.2f`). Los datos en el archivo de texto se pueden ver con un editor, o posteriormente mediante un programa diseñado para leer el archivo (sección 15.4.2).

Cuando se ejecutan las líneas 53 a 55, si se cierra el objeto `Formatter` se lanza una excepción tipo **`FormatterClosedException`**. Esta excepción se maneja en la línea 56. [*Nota*: también puede enviar datos a un archivo de texto mediante la clase **`java.io.PrintWriter`**, la cual cuenta también con los métodos `format` y `printf` para imprimir datos con formato].

En las líneas 72 a 76 se declara el método `cerrarArchivo`, el cual cierra el objeto `Formatter` y el archivo de salida subyacente. En la línea 75 se cierra el objeto mediante una llamada simple al método **`close`**. Si el método `close` no se llama en forma explícita, el sistema operativo por lo general cierra el archivo cuando el programa termina de ejecutarse; éste es un ejemplo de las "tareas de mantenimiento" del sistema operativo. Sin embargo, siempre debemos cerrar un archivo en forma explícita cuando ya no lo necesitemos.

Resultados de ejemplo

Los datos de ejemplos para esta aplicación se muestran en la figura 15.5. En estos resultados, el usuario introduce información para cinco cuentas y luego el fin de archivo para indicar que terminó de introducir datos. Los resultados de ejemplo no muestran cómo aparecen realmente los registros de datos en el archivo. Para verificar que el archivo se haya creado exitosamente, en la siguiente sección presentamos un programa que lee el archivo e imprime su contenido. Como es un archivo de texto, también puede verificar la información con sólo abrir el archivo en un editor de texto.

Datos de ejemplo			
100	Bob	Blue	24.98
200	Steve	Green	-345.67
300	Pam	White	0.00
400	Sam	Red	-42.16
500	Sue	Yellow	224.62

Fig. 15.5 | Datos de ejemplo para el programa de la figura 15.3.

15.4.2 Cómo leer datos de un archivo de texto de acceso secuencial

Los datos se almacenan en archivos para poder procesarlos según sea necesario. En la sección 15.4.1 demostramos cómo crear un archivo para acceso secuencial. Esta sección muestra cómo leer los datos en forma secuencial desde un archivo de texto. Demostraremos cómo puede utilizarse la clase `Scanner` para recibir datos de un archivo, en vez del teclado. La aplicación (figura 15.6) lee registros del archivo "`clientes.txt`" creado por la aplicación de la sección 15.4.1 y muestra el contenido de los registros. En la línea 13 se declara un objeto `Scanner`, que se utilizará para obtener los datos de entrada del archivo.

```
1  // Fig. 15.6: LeerArchivoTexto.java
2  // Este programa lee un archivo de texto y muestra cada registro.
3  import java.io.IOException;
4  import java.lang.IllegalStateException;
5  import java.nio.file.Files;
6  import java.nio.file.Path;
7  import java.nio.file.Paths;
8  import java.util.NoSuchElementException;
9  import java.util.Scanner;
10
11 public class LeerArchivoTexto
12 {
13    private static Scanner entrada;
14
15    public static void main(String[] args)
16    {
17       abrirArchivo();
18       leerRegistros();
19       cerrarArchivo();
20    }
21
22    // abre el archivo clientes.txt
23    public static void abrirArchivo()
24    {
25       try
26       {
27          entrada = new Scanner(Paths.get("clientes.txt"));
28       }
```

Fig. 15.6 | Lectura de un archivo secuencial mediante un objeto `Scanner` (parte 1 de 2).

```
29          catch (IOException ioException)
30          {
31             System.err.println("Error al abrir el archivo. Terminando.");
32             System.exit(1);
33          }
34       }
35
36       // lee registro del archivo
37       public static void leerRegistros()
38       {
39          System.out.printf("%-10s%-12s%-12s%10s%n", "Cuenta",
40             "Primer nombre", "Apellido paterno", "Saldo");
41
42          try
43          {
44             while (entrada.hasNext) // mientras haya más qué leer
45             {
46                // muestra el contenido del registro
47                System.out.printf("%-10d%-12s%-12s%10.2f%n", entrada.nextInt(),
48                   entrada.next(), entrada.next(), entrada.nextDouble());
49             }
50          }
51          catch (NoSuchElementException elementException)
52          {
53             System.err.println("El archivo no esta bien formado. Terminando.");
54          }
55          catch (IllegalStateException stateException)
56          {
57             System.err.println("Error al leer del archivo. Terminando.");
58          }
59       } // fin del método leerRegistros
60
61       // cierra el archivo y termina la aplicación
62       public static void cerrarArchivo()
63       {
64          if (entrada != null)
65             entrada.close();
66       }
67    } // fin de la clase LeerArchivoTexto
```

```
Cuenta     Primer nombre     Apellido paterno     Saldo
100        Bob               Blue                 24.98
200        Steve             Green                -345.67
300        Pam               White                0.00
400        Sam               Red                  -42.16
500        Sue               Yellow               224.62
```

Fig. 15.6 | Lectura de un archivo secuencial mediante un objeto `Scanner` (parte 2 de 2).

El método `abrirArchivo` (líneas 23 a 34) abre el archivo en modo de lectura, creando una instancia de un objeto `Scanner` en la línea 27. Pasamos un objeto `Path` al constructor, el cual especifica que el objeto `Scanner` leerá datos del archivo "`clientes.txt`" ubicado en el directorio desde el que se ejecuta la aplicación. Si no puede encontrarse el archivo, ocurre una excepción tipo `IOException`. La excepción se maneja en las líneas 29 a 33.

El método `leerRegistros` (líneas 37 a 59) lee y muestra registros del archivo. En las líneas 39 a 40 se muestran encabezados para las columnas, dentro de los resultados de la aplicación. En las líneas 44 a 49 se leen y muestran datos del archivo hasta llegar al *marcador de fin de archivo* (en cuyo caso, el método `hasNext` devolverá `false` en la línea 44). En las líneas 47 a 48 se utilizan los métodos `nextInt`, `next` y `nextDouble` de `Scanner` para recibir un `int` (el número de cuenta), dos objetos `String` (el primer nombre y el apellido paterno) y un valor `double` (el saldo). Cada registro es una línea de datos en el archivo. Si la información en el archivo no está bien formada (por ejemplo, que haya un apellido paterno en donde debe haber un saldo), se produce una excepción tipo `NoSuchElementException` al momento de introducir el registro. Esta excepción se maneja en las líneas 51 a 54. Si el objeto `Scanner` se cerró antes de introducir los datos, se produce una excepción tipo `IllegalStateException` (que se maneja en las líneas 55 a 58). Observe en la cadena de formato de la línea 47 que el número de cuenta, el primer nombre y el apellido paterno están justificados a la izquierda, mientras que el saldo está justificado a la derecha y se imprime con dos dígitos de precisión. Cada iteración del ciclo introduce una línea de texto del archivo de texto, la cual representa un registro. En las líneas 62 a 66 se define el método `cerrarArchivo`, el cual cierra el objeto `Scanner`.

15.4.3 Ejemplo práctico: un programa de solicitud de crédito

Para obtener datos secuencialmente de un archivo, por lo general los programas empiezan desde el principio del archivo y leen *todos* los datos en forma consecutiva hasta encontrar la información deseada. Durante la ejecución de un programa podría ser necesario procesar el archivo secuencialmente varias veces (desde el principio del archivo). La clase `Scanner` *no* proporciona la habilidad de reposicionarse hasta el principio del archivo. Si es necesario leer el archivo de nuevo, el programa debe *cerrar* el archivo y *volver a abrirlo*.

El programa de las figuras 15.7 a 15.8 permite a un gerente de créditos obtener listas de clientes con *saldos de cero* (es decir, los clientes que no deben dinero), *saldos con crédito* (es decir, los clientes a quienes la compañía les debe dinero) y *saldos con débito* (es decir, los clientes que deben dinero a la compañía por los bienes y servicios recibidos en el pasado). Un saldo con crédito es un monto *negativo*, y un saldo con débito es un monto *positivo*.

La enumeración `OpcionMenu`

Empezamos por crear un tipo `enum` (figura 15.7) para definir las distintas opciones del menú que tendrá el gerente de créditos; esto es obligatorio si necesita proporcionar valores específicos para las constantes `enum`. Las opciones y sus valores se listan en las líneas 7 a 10.

```
1  // Fig. 15.7: OpcionMenu.java
2  // Enumeración para las opciones del programa de consulta de crédito.
3
4  public enum OpcionMenu
5  {
6     // declara el contenido del tipo enum
7     SALDO_CERO(1),
8     SALDO_CREDITO(2),
9     SALDO_DEBITO(3),
10    FIN(4);
11
12    private final int valor; // opción actual del menú
13
```

Fig. 15.7 | Enumeración para las opciones del menú del programa de consulta de crédito (parte 1 de 2).

```
14      // constructor
15      private OpcionMenu(int valor)
16      {
17         this.valor = valor;
18      }
19   } // fin del tipo enum OpcionMenu
```

Fig. 15.7 | Enumeración para las opciones del menú del programa de consulta de crédito (parte 2 de 2).

La clase `ConsultaCredito`

La figura 15.18 contiene la funcionalidad para el programa de consulta de crédito. Este programa muestra un menú de texto y permite al gerente de créditos introducir una de tres opciones para obtener información sobre un crédito:

- La opción 1 (`SALDO_CERO`) muestra las cuentas con saldos de cero.
- La opción 2 (`SALDO_CREDITO`) muestra las cuentas con saldos con crédito.
- La opción 3 (`SALDO_DEBITO`) muestra las cuentas con saldos con débito.
- La opción 4 (`FIN`) termina la ejecución del programa.

```
1  // Fig. 15.8: ConsultaCredito.java
2  // Este programa lee un archivo secuencialmente y muestra su
3  // contenido con base en el tipo de cuenta que solicita el usuario
4  // (saldo con crédito, saldo con débito o saldo de cero).
5  import java.io.IOException;
6  import java.lang.IllegalStateException;
7  import java.nio.file.Paths;
8  import java.util.NoSuchElementException;
9  import java.util.Scanner;
10
11 public class ConsultaCredito
12 {
13    private final static OpcionMenu[] opciones = OpcionMenu.values();
14
15    public static void main(String[] args)
16    {
17       // obtiene la solicitud del usuario (saldo de cero, con crédito o con débito)
18       OpcionMenu tipoCuenta = obtenerSolicitud();
19
20       while (tipoCuenta != OpcionMenu.FIN)
21       {
22          switch (tipoCuenta)
23          {
24             case SALDO_CERO:
25                System.out.printf("%nCuentas con saldos de cero:%n");
26                break;
27             case SALDO_CREDITO:
28                System.out.printf("%nCuentas con saldos con credito:%n");
29                break;
```

Fig. 15.8 | Programa de consulta de crédito (parte 1 de 4).

```
30              case SALDO_DEBITO:
31                 System.out.printf("%nCuentas con saldos con debito:%n");
32                 break;
33           }
34
35           leerRegistros(tipoCuenta);
36           tipoCuenta = obtenerSolicitud(); // obtiene la solicitud del usuario
37        }
38     }
39
40     // obtiene la solicitud del usuario
41     private static OpcionMenu obtenerSolicitud()
42     {
43        int solicitud = 4;
44
45        // muestra opciones de solicitud
46        System.out.printf("%nEscriba solicitud%n%s%n%s%n%s%n%s%n",
47           " 1 - Lista de cuentas con saldos de cero",
48           " 2 - Lista de cuentas con saldos con credito",
49           " 3 - Lista de cuentas con saldos con debito",
50           " 4 - Terminar programa");
51
52        try
53        {
54           Scanner entrada = new Scanner(System.in);
55
56           do // recibe solicitud del usuario
57           {
58              System.out.printf("%n? ");
59              solicitud = entrada.nextInt();
60           } while ((solicitud < 1) || (solicitud > 4));
61        }
62        catch (NoSuchElementException noSuchElementException)
63        {
64           System.err.println("Entrada invalida. Terminando.");
65        }
66
67        return opciones[solicitud - 1]; // devuelve valor de enum para la opción
68     }
69
70     // lee los registros del archivo y muestra sólo los registros del tipo apropiado
71     private static void leerRegistros(OpcionMenu tipoCuenta)
72     {
73        // abre el archivo y procesa el contenido
74        try (Scanner entrada = new Scanner(Paths.get("clientes.txt")))
75        {
76           while (entrada.hasNext()) // más datos a leer
77           {
78              int numeroCuenta = entrada.nextInt();
79              String primerNombre = entrada.next();
80              String apellidoPaterno = entrada.next();
81              double saldo = entrada.nextDouble();
82
```

Fig. 15.8 | Programa de consulta de crédito (parte 2 de 4).

```
83              // si el tipo de cuenta es apropiado, muestra el registro
84              if (debeMostrar(tipoCuenta, saldo))
85                 System.out.printf("%-10d%-12s%-12s%10.2f%n", numeroCuenta,
86                    primerNombre, apellidoPaterno, saldo);
87              else
88                 entrada.nextLine(); // descarta el resto del registro actual
89           }
90        }
91        catch (NoSuchElementException |
92           IllegalStateException | IOException e)
93        {
94           System.err.println("Error al procesar el archivo. Terminando.");
95           System.exit(1);
96        }
97     } // fin del método leerRegistros
98
99     // usa el tipo de registro para determinar si el registro debe mostrarse
100    private static boolean debeMostrar(
101       OpcionMenu tipoCuenta, double saldo)
102    {
103       if ((tipoCuenta == OpcionMenu.SALDO_CREDITO) && (saldo < 0))
104          return true;
105       else if ((tipoCuenta == OpcionMenu.SALDO_DEBITO) && (saldo > 0))
106          return true;
107       else if ((tipoCuenta == OpcionMenu.SALDO_CERO) && (saldo == 0))
108          return true;
109
110       return false;
111    }
112 } // fin de la clase ConsultaCredito
```

```
Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Terminar programa

? 1

Cuentas con saldos de cero:
300       Pam         White             0.00

Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Terminar programa

? 2

Cuentas con saldos con credito:
200       Steve       Green          -345.67
400       Sam         Red             -42.16
```

Fig. 15.8 | Programa de consulta de crédito (parte 3 de 4).

```
Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Terminar programa

? 3

Cuentas con saldos con debito:
100       Bob         Blue             24.98
500       Sue         Yellow          224.62

Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Terminar programa

? 4
```

Fig. 15.8 | Programa de consulta de crédito (parte 4 de 4).

Para recolectar la información de los registros, se lee todo el archivo completo y se determina si cada uno de los registros cumple o no con los criterios para el tipo de cuenta seleccionado. En la línea 18 en `main` se hace una llamada al método `obtenerSolicitud` (líneas 41 a 68) para mostrar las opciones del menú, se traduce el número introducido por el usuario en un objeto `OpcionMenu` y se almacena el resultado en la variable `OpcionMenu` llamada `tipoCuenta`. En las líneas 20 a 37 se itera hasta que el usuario especifique que el programa debe terminar. En las líneas 22 a 33 se muestra un encabezado para imprimir el conjunto actual de registros en la pantalla. En la línea 35 se hace una llamada al método `leerRegistros` (líneas 71 a 97), el cual itera a través del archivo y lee todos los registros.

El método `leerRegistros` usa una instrucción `try` con recursos (que se presentó en la sección 11.12) para crear un objeto `Scanner` que abre el archivo en modo de lectura (línea 74). Recuerde que `try` con recursos cierra sus recursos cuando el bloque `try` termina con éxito o debido a una excepción. Cada vez que se haga una llamada a `leerRegistros`, el archivo se abrirá en modo de lectura con un nuevo objeto `Scanner` para que podamos leer de nuevo desde el principio del archivo. En las líneas 78 a 81 se lee un registro. En la línea 84 se hace una llamada al método `debeMostrar` (líneas 100 a 111), para determinar si el registro actual cumple con el tipo de cuenta solicitado. Si `debeMostrar` devuelve `true`, el programa muestra la información de la cuenta. Al llegar al *marcador de fin de archivo*, el ciclo termina y la instrucción `try` con recursos cierra el objeto `Scanner` junto con el archivo. Una vez que se hayan leído todos los registros, el control regresa al método `main` y se hace otra vez una llamada al método `obtenerSolicitud` (línea 36) para obtener la siguiente opción de menú del usuario.

15.4.4 Actualización de archivos de acceso secuencial

En muchos archivos secuenciales los datos no se pueden modificar sin el riesgo de destruir otros datos en el archivo. Por ejemplo, si el nombre "`White`" tuviera que cambiarse a "`Worthington`", el nombre anterior no podría simplemente sobrescribirse ya que el nuevo nombre requiere más espacio. El registro para `White` se escribió en el archivo como

```
300 Pam White 0.00
```

Si el registro se sobrescribe empezando en la misma ubicación en el archivo que utiliza el nuevo nombre, el registro será

```
300 Pam Worthington 0.00
```

El nuevo registro es más extenso (tiene más caracteres) que el registro original. "`Worthington`" sobrescribiría el valor "`0.00`" en el registro actual y los caracteres más allá de la segunda "`o`" en "`Worthington`" sobrescribirán el principio del siguiente registro secuencial en el archivo. El problema aquí es que los campos en un archivo de texto (y por ende, los registros) pueden variar en tamaño. Por ejemplo, 7, 14, -117, 2074 y 27383 son todos valores `int` almacenados en el mismo número de bytes (4) internamente, pero son campos con distintos tamaños cuando se escriben en un archivo como texto. Por lo tanto, los registros en un archivo de acceso secuencial comúnmente no se actualizan por partes. En vez de ello, se sobrescribe todo el archivo. Para realizar el cambio anterior, los registros que se encuentran antes de `300 Pam White 0.00` se copian a un nuevo archivo, luego se escribe el nuevo registro (que puede tener un tamaño distinto al que está sustituyendo) y se copian los registros después de `300 Pam White 0.00` al nuevo archivo. No es muy conveniente actualizar sólo un registro, pero resulta razonable si una porción sustancial de los registros necesita actualización.

15.5 Serialización de objetos

En la sección 15.4 demostramos cómo escribir los campos individuales de un registro en un archivo como texto y cómo leer esos campos desde un archivo. Cuando los datos se enviaban al disco, se perdía cierta información como el tipo de cada valor. Por ejemplo, si se lee el valor "`3`" de un archivo, no hay forma de saber si el valor proviene de un `int`, un `String` o un `double`. En un disco sólo tenemos los datos, no la información sobre los tipos.

Algunas veces es necesario leer o escribir un objeto en un archivo o a través de una conexión de red. Java cuenta con la **serialización de objetos** para estos fines. Un **objeto serializado** es un objeto que se representa como una secuencia de bytes, la cual incluye los datos del objeto, así como información acerca del tipo del objeto y los tipos de los datos almacenados en el mismo. Una vez que se escribe un objeto serializado en un archivo, se puede leer desde ese archivo y **deserializarse**; es decir, la información del tipo y los bytes que representan al objeto y sus datos se puede utilizar para recrear el objeto en memoria.

Las clases `ObjectInputStream` y `ObjectOutputStream`

Las clases **`ObjectInputStream`** y **`ObjectOutputStream`** (paquete `java.io`), que implementan respectivamente a las interfaces **`ObjectInput`** y **`ObjectOutput`**, permiten leer y escribir objetos completos desde o en un flujo (posiblemente un archivo). Para utilizar la serialización con los archivos, inicializamos los objetos `ObjectInputStream` y `ObjectOutputStream` con objetos flujo que pueden leer y escribir información desde y hacia los archivos. La acción de inicializar de esta forma objetos flujo con otros objetos flujo se conoce algunas veces como **envoltura**, ya que el nuevo objeto flujo que se va a crear "envuelve" al objeto flujo especificado como un argumento del constructor.

Las clases `ObjectInputStream` y `ObjectOutputStream` simplemente leen y escriben la representación basada en bytes de los objetos; no saben de dónde leer los bytes o en dónde escribirlos. El objeto flujo que pasamos al constructor de `ObjectInputStream` recibe la representación basada en bytes del objeto que el `ObjectOutputStream` produce, y escribe los bytes en el destino especificado (por ejemplo, un archivo o una conexión en red).

Las interfaces `ObjectOutput` y `ObjectInput`

La interfaz `ObjectOutput` contiene el método **`writeObject`**, el cual toma un objeto `Object` como un argumento y escribe su información en un objeto `OutputStream`. Una clase que implementa a la interfaz `ObjectOutput` (como `ObjectOutputStream`) declara este método y se asegura de que el objeto que se va a producir implemente la interfaz `Serializable` (que veremos en breve). De manera correspondiente, la interfaz `ObjectInput` contiene el método **`readObject`** que lee y devuelve una referencia a un objeto `Object` de un objeto `InputStream`. Una vez que se lee un objeto, su referencia puede convertirse en el tipo actual del objeto. Como veremos en el capítulo 28 (en inglés, en el sitio web del libro), las aplicaciones que se comunican a través de una red (como Internet) también pueden transmitir objetos completos a través de la red.

15.5.1 Creación de un archivo de acceso secuencial mediante el uso de la serialización de objetos

En esta sección y en la sección 15.5.2 vamos a crear y manipular archivos de acceso secuencial, usando la serialización de objetos. La serialización de objetos que mostraremos aquí se realiza mediante flujos basados en bytes, de manera que los archivos secuenciales que se creen y manipulen serán *archivos binarios*. Recuerde que por lo general los archivos binarios no se pueden ver en los editores de texto estándar. Por esta razón, escribimos una aplicación independiente que sabe cómo leer y mostrar objetos serializados. Empezaremos por crear y escribir objetos serializados en un archivo de acceso secuencial. El ejemplo es similar al de la sección 15.4, por lo que sólo nos enfocaremos en las nuevas características.

Definición de la clase `Cuenta`

Para empezar, vamos a definir la clase `Cuenta` (figura 15.9), la cual encapsula la información de los registros de los clientes que utilizan los ejemplos de serialización. Estos ejemplos y la clase `Cuenta` se encuentran en el directorio `AppsSerializacion` con los ejemplos del capítulo. Esto permite que ambos ejemplos usen la clase `Cuenta`, ya que sus archivos están definidos en el mismo paquete predeterminado. La clase `Cuenta` contiene las variables de instancia `private cuenta`, `primerNombre`, `apellidoPaterno` y `saldo` (líneas 7 a 10), además de métodos *establecer* y *obtener* para acceder a estas variables de instancia. Aunque los métodos *establecer* no validan los datos en este ejemplo, en un sistema de "nivel industrial" sí deben hacerlo. La clase `Cuenta` implementa a la interfaz **`Serializable`** (línea 5), la cual permite *serializar* y *deserializar* los objetos de esta clase con objetos `ObjectOutputStream` y `ObjectInputStream`, respectivamente. La interfaz `Serializable` es una **interfaz de marcado**. Dicha interfaz *no* contiene métodos. Una clase que implementa a `Serializable` se *marca* como objeto `Serializable`. Esto es importante ya que un objeto `ObjectOutputStream` *no* enviará un objeto como salida a menos que *sea un* objeto `Serializable`, lo cual es el caso para cualquier objeto de una clase que implemente a `Serializable`.

```
1  // Fig. 15.9: Cuenta.java
2  // La clase serializable Cuenta para almacenar registros como objetos.
3  import java.io.Serializable;
4
5  public class Cuenta implements Serializable
6  {
7     private int cuenta;
8     private String primerNombre;
9     private String apellidoPaterno;
10    private double saldo;
11
```

Fig. 15.9 | La clase `Cuenta` para los objetos serializables (parte 1 de 3).

```
12    // inicializa un objeto Cuenta con valores predeterminados
13    public Cuenta()
14    {
15       this(0, "", "", 0.0); // llama a otro constructor
16    }
17
18    // inicializa un objeto Cuenta con los valores proporcionados
19    public Cuenta(int cuenta, String primerNombre,
20       String apellidoPaterno, double saldo)
21    {
22       this.cuenta = cuenta;
23       this.primerNombre = primerNombre;
24       this.apellidoPaterno = apellidoPaterno;
25       this.saldo = saldo;
26    }
27
28    // establece el número de cuenta
29    public void establecerCuenta(int cuenta)
30    {
31       this.cuenta = cuenta;
32    }
33
34    // obtiene el número de cuenta
35    public int obtenerCuenta()
36    {
37       return cuenta;
38    }
39
40    // establece el primer nombre
41    public void establecerPrimerNombre(String primerNombre)
42    {
43       this.primerNombre = primerNombre;
44    }
45
46    // obtiene el primer nombre
47    public String obtenerPrimerNombre()
48    {
49       return primerNombre;
50    }
51
52    // establece el apellido paterno
53    public void establecerApellidoPaterno(String apellidoPaterno)
54    {
55       this.apellidoPaterno = apellidoPaterno;
56    }
57
58    // obtiene el apellido paterno
59    public String obtenerApellidoPaterno()
60    {
61       return apellidoPaterno;
62    }
63
```

Fig. 15.9 | La clase `Cuenta` para los objetos serializables (parte 2 de 3).

```
64      // establece el saldo
65      public void establecerSaldo(double saldo)
66      {
67         this.saldo = saldo;
68      }
69
70      // obtiene el saldo
71      public double obtenerSaldo()
72      {
73         return saldo;
74      }
75   } // fin de la clase Cuenta
```

Fig. 15.9 | La clase `Cuenta` para los objetos serializables (parte 3 de 3).

En una clase `Serializable`, cada variable de instancia debe ser `Serializable`. Cualquier variable de instancia que no sea `Serializable` debe declararse como **`transient`** para indicar que debe ignorarse durante el proceso de serialización. *De manera predeterminada, todas las variables de tipos primitivos son serializables*. Para las variables de tipos de referencias, hay que verificar la documentación de la clase (y posiblemente de sus superclases) para asegurar que el tipo sea `Serializable`. Por ejemplo, los objetos `String` son `Serializable`. De manera predeterminada, los arreglos *son* serializables; no obstante, en un arreglo de tipo por referencia los objetos referenciados tal vez *no* lo sean. La clase `Cuenta` contiene los miembros de datos `private` llamados `cuenta`, `primerNombre`, `apellidoPaterno` y `saldo`; todos ellos son `Serializable`. Esta clase también proporciona métodos `public` *establecer* y *obtener* para acceder a los campos `private`.

Escritura de objetos serializados en un archivo de acceso secuencial

Ahora hablaremos sobre el código que crea el archivo de acceso secuencial (figura 15.10). Aquí nos concentraremos sólo en los nuevos conceptos. Para abrir el archivo, en la línea 27 se hace una llamada al método `static` **`newOutputStream`** de `Files`, el cual recibe un objeto `Path` que especifica el archivo a abrir y si éste existe devuelve un objeto `OutputStream` que puede usarse para escribir en el archivo. Los archivos existentes que se abren de esta forma en modo de salida, se *truncan*. No hay una extensión de nombre de archivo estándar para los archivos que almacenan objetos serializados; en este caso elegimos la extensión `.ser`.

```
1  // Fig. 15.10: CrearArchivoSecuencial.java
2  // Escritura de objetos en forma secuencial a un archivo, con la clase
      ObjectOutputStream.
3  import java.io.IOException;
4  import java.io.ObjectOutputStream;
5  import java.nio.file.Files;
6  import java.nio.file.Paths;
7  import java.util.NoSuchElementException;
8  import java.util.Scanner;
9
10 public class CrearArchivoSecuencial
11 {
12    private static ObjectOutputStream salida; // envía los datos a un archivo
13
```

Fig. 15.10 | Archivo secuencial creado mediante `ObjectOutputStream` (parte 1 de 3).

```
14    public static void main(String[] args)
15    {
16       abrirArchivo();
17       agregarRegistros();
18       cerrarArchivo();
19    }
20
21    // abre el archivo clientes.ser
22    public static void abrirArchivo()
23    {
24       try
25       {
26          salida = new ObjectOutputStream(
27             Files.newOutputStream(Paths.get("clientes.ser")));
28       }
29       catch (IOException ioException)
30       {
31          System.err.println("Error al abrir el archivo. Terminando.");
32          System.exit(1); // termina el programa
33       }
34    }
35
36    // agrega registros al archivo
37    public static void agregarRegistros()
38    {
39       Scanner entrada = new Scanner(System.in);
40
41       System.out.printf("%s%n%s%n? ",
42          "Escriba numero de cuenta, primer nombre, apellido paterno y saldo.",
43          "Escriba el indicador de fin de archivo para terminar la entrada.");
44
45       while (entrada.hasNext()) // itera hasta el indicador de fin de archivo
46       {
47          try
48          {
49             // crea nuevo registro; este ejemplo asume una entrada válida
50             Cuenta registro = new Cuenta(entrada.nextInt(),
51                entrada.next(), entrada.next(), entrada.nextDouble());
52
53             // serializa el objeto registro en el archivo
54             salida.writeObject(registro);
55          }
56          catch (NoSuchElementException elementException)
57          {
58             System.err.println("Entrada invalida. Intente de nuevo.");
59             entrada.nextLine(); // descarta la entrada para que el usuario pueda
                                   intentar de nuevo
60          }
61          catch (IOException ioException)
62          {
63             System.err.println("Error al escribir en el archivo. Terminando.");
64             break;
65          }
```

Fig. 15.10 | Archivo secuencial creado mediante `ObjectOutputStream` (parte 2 de 3).

```
66
67             System.out.print("? ");
68          }
69       }
70
71       // cierra el archivo y termina la aplicación
72       public static void cerrarArchivo()
73       {
74          try
75          {
76             if (salida != null)
77                salida.close();
78          }
79          catch (IOException ioException)
80          {
81             System.err.println("Error al cerrar el archivo. Terminando.");
82          }
83       }
84    } // fin de la clase CrearArchivoSecuencial
```

```
Escriba numero de cuenta, primer nombre, apellido paterno y saldo.
Escriba el indicador de fin de archivo para terminar la entrada.
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

Fig. 15.10 | Archivo secuencial creado mediante `ObjectOutputStream` (parte 3 de 3).

La clase `OutputStream` cuenta con métodos para escribir arreglos tipo `byte` y objetos `byte` individuales en un archivo, pero nosotros queremos escribir *objetos* en un archivo. Por esta razón, en las líneas 26 y 27 pasamos el objeto `InputStream` al constructor de la clase `OutputStream` que *envuelve* el objeto `OutputStream` en un objeto `ObjectOutputStream`. El objeto `ObjectOutputStream` utiliza al objeto `OutputStream` para escribir en el archivo los bytes que representan objetos enteros. En las líneas 26 y 27 se podría lanzar una excepción tipo **`IOException`** si ocurriera un problema al abrir el archivo (por ejemplo, cuando se abre un archivo para escribir en una unidad de disco con espacio insuficiente o cuando se abre un archivo de sólo lectura para escribir datos). En estos casos el programa muestra un mensaje de error (líneas 29 a 33). Si no ocurre una excepción, el archivo se abre y se puede utilizar la variable `salida` para escribir objetos en el archivo.

Este programa asume que los datos se introducen de manera correcta y en el orden de número de registro apropiado. El método `agregarRegistros` (líneas 37 a 69) realiza la operación de escritura. En las líneas 50 y 51 se crea un objeto `Cuenta` a partir de los datos introducidos por el usuario. En la línea 54 se hace una llamada al método `writeObject` de `ObjectOutputStream` para escribir el objeto `registro` en el archivo de salida. Observe que sólo se requiere una instrucción para escribir *todo* el objeto.

El método `cerrarArchivo` (líneas 72 a 83) llama al método **`close`** de `ObjectOutputStream` en `salida` para cerrar el objeto `ObjectOutputStream` *y* su objeto `OutputStream` subyacente. La llamada al método `close` está dentro de un bloque `try` ya que `close` lanza una excepción `IOException` cuando el archivo no se puede cerrar en forma apropiada. Al utilizar flujos *envueltos*, si se cierra el flujo exterior *también* se cierra el flujo envuelto.

En la ejecución de ejemplo para el programa de la figura 15.10 introdujimos información para cinco cuentas; la misma información que se muestra en la figura 15.5. El programa no muestra cómo es que aparecen en realidad los registros en el archivo. Recuerde que ahora estamos usando *archivos binarios* que no pueden ser leídos por los seres humanos. Para verificar que el archivo se haya creado con éxito, la siguiente sección presenta un programa para leer el contenido del mismo.

15.5.2 Lectura y deserialización de datos de un archivo de acceso secuencial

En la sección anterior mostramos cómo crear un archivo para acceso secuencial usando la serialización de objetos. En esta sección veremos cómo *leer datos serializados* de un archivo, en forma secuencial.

El programa de la figura 15.11 lee registros de un archivo creado por el programa de la sección 15.5.1 y muestra el contenido. El programa abre el archivo en modo de entrada mediante una llamada al método `static newInputStream` de `Files`, que recibe un objeto `Path` en el que se especifica el archivo a abrir y, si éste existe, devuelve un objeto `InputStream` que puede usarse para leer datos del archivo. En la figura 15.10 escribimos objetos al archivo usando un objeto `ObjectOutputStream`. Los datos se deben leer del archivo en el mismo formato en el que se escribió. Por lo tanto, utilizamos un objeto `ObjectInputStream` *envuelto* alrededor de un objeto `InputStream` (líneas 26 y 27). Si no ocurren excepciones al abrir el archivo, podemos usar la variable `entrada` para leer objetos del archivo.

```
1  // Fig. 15.11: LeerArchivoSecuencial.java
2  // Leer un archivo de objetos en forma secuencial con ObjectInputStream
3  // y mostrar cada registro en pantalla.
4  import java.io.EOFException;
5  import java.io.IOException;
6  import java.io.ObjectInputStream;
7  import java.nio.file.Files;
8  import java.nio.file.Paths;
9
10 public class LeerArchivoSecuencial
11 {
12    private static ObjectInputStream entrada;
13
14    public static void main(String[] args)
15    {
16       abrirArchivo();
17       leerRegistros();
18       cerrarArchivo();
19    }
20
21    // permite al usuario seleccionar el archivo a abrir
22    public static void abrirArchivo()
23    {
24       try // abre el archivo
25       {
26          entrada = new ObjectInputStream(
27             Files.newInputStream(Paths.get("clientes.ser")));
28       }
29       catch (IOException ioException)
30       {
```

Fig. 15.11 | Lectura de un archivo de objetos en forma secuencial mediante `ObjectInputStream` y muestra de cada registro en pantalla (parte 1 de 3).

```
31            System.err.println("Error al abrir el archivo.");
32            System.exit(1);
33         }
34      }
35
36      // lee el registro del archivo
37      public static void leerRegistros()
38      {
39         System.out.printf("%-10s%-12s%-12s%10s%n", "Cuenta",
40            "Primer nombre", "Apellido paterno", "Saldo");
41
42         try
43         {
44            while (true) // itera hasta que haya una EOFException
45            {
46               Cuenta registro = (Cuenta) entrada.readObject();
47
48               // muestra el contenido del registro
49               System.out.printf("%-10d%-12s%-12s%10.2f%n",
50                  registro.obtenerCuenta(), registro.obtenerPrimerNombre(),
51                  registro.obtenerApellidoPaterno(), registro.obtenerSaldo());
52            }
53         }
54         catch (EOFException endOfFileException)
55         {
56            System.out.printf("%No hay mas registros%n");
57         }
58         catch (ClassNotFoundException classNotFoundException)
59         {
60            System.err.println("Tipo de objeto invalido. Terminando.");
61         }
62         catch (IOException ioException)
63         {
64            System.err.println("Error al leer del archivo. Terminando.");
65         }
66      } // fin del método leerRegistros
67
68      // cierra el archivo y termina la aplicación
69      public static void cerrarArchivo()
70      {
71         try
72         {
73            if (entrada != null)
74               entrada.close();
75         }
76         catch (IOException ioException)
77         {
78            System.err.println("Error al cerrar el archivo. Terminando.");
79            System.exit(1);
80         }
81      }
82   } // fin de la clase LeerArchivoSecuencial
```

Fig. 15.11 | Lectura de un archivo de objetos en forma secuencial mediante `ObjectInputStream` y muestra de cada registro en pantalla (parte 2 de 3).

```
Cuenta      Primer nombre     Apellido paterno     Saldo
100         Bob               Blue                 24.98
200         Steve             Green                -345.67
300         Pam               White                0.00
400         Sam               Red                  -42.16
500         Sue               Yellow               224.62

No hay mas registros
```

Fig. 15.11 | Lectura de un archivo de objetos en forma secuencial mediante `ObjectInputStream` y muestra de cada registro en pantalla (parte 3 de 3).

El programa lee registros del archivo en el método `leerRegistros` (líneas 37 a 66). En la línea 46 se hace una llamada al método `readObject` de `ObjectInputStream` para leer un objeto `Object` del archivo. Para utilizar los métodos específicos de `Cuenta` realizamos una *conversión descendente* en el objeto `Object` devuelto, al tipo `Cuenta`. Si se hace un intento por leer más allá del fin del archivo, el método `readObject` lanza una excepción tipo `EOFException` (que se procesa en las líneas 54 a 57). Si no se puede localizar la clase para el objeto que se está leyendo, el método `readObject` lanza una excepción `ClassNotFoundException`. Esto podría ocurrir si se accede al archivo en una computadora que no tenga esa clase.

Observación de ingeniería de software 15.1

En esta sección se presentó la serialización de objetos y se demostraron las técnicas básicas de serialización de objetos. La serialización es un tema profundo con muchas trampas y obstáculos. Antes de implementar la serialización en aplicaciones de uso industrial, lea con cuidado la documentación de Java en línea correspondiente a la serialización de objetos.

15.6 Apertura de archivos con `JFileChooser`

La clase `JFileChooser` muestra un cuadro de diálogo que permite al usuario seleccionar con facilidad archivos o directorios. Para demostrar este cuadro de diálogo mejoramos el ejemplo de la sección 15.3, como se muestra en las figuras 15.12 y 15.13. El ejemplo ahora contiene una interfaz gráfica de usuario pero sigue mostrando los mismos datos. El constructor llama al método `analizarRuta` en la línea 24. Después, este método llama al método `obtenerRutaArchivoODirectorio` en la línea 31 para obtener un objeto `Path` que representa el archivo o directorio seleccionado.

El método `obtenerRutaArchivoODirectorio` (líneas 71 a 85 de la figura 15.12) crea un objeto `JFileChooser` (línea 74). En las líneas 75 y 76 se hace una llamada al método `setFileSelectionMode` para especificar lo que el usuario puede seleccionar del objeto `selectorArchivos`. Para este programa utilizamos la constante `static` `FILES_AND_DIRECTORIES` de `JFileChooser` para indicar que pueden seleccionarse archivos y directorios. Otras constantes `static` son `FILES_ONLY` (sólo archivos, el valor predeterminado) y `DIRECTORIES_ONLY` (sólo directorios).

En la línea 77 se hace una llamada al método `showOpenDialog` para mostrar el cuadro de diálogo `JFileChooser` llamado **Abrir**. El argumento `this` especifica la ventana padre del cuadro de diálogo `JFileChooser`, la cual determina la posición del cuadro de diálogo en la pantalla. Si se pasa `null`, el cuadro de diálogo se muestra en el centro de la pantalla; en caso contrario, el cuadro de diálogo se centra sobre la ventana de la aplicación (lo cual se especifica mediante el argumento `this`). Un cuadro de diálogo `JFileChooser` es un *cuadro de diálogo modal* que no permite al usuario interactuar con cualquier otra ventana en el programa, sino hasta que el usuario cierre el diálogo. El usuario selecciona la unidad, el nombre del directorio o archivo, y después hace clic en **Abrir**. El método `showOpenDialog` devuelve un entero, el cual especifica qué botón (**Abrir** o **Cancelar**) oprimió el usuario para cerrar el cuadro de diálogo. En la línea 48 se evalúa si el usuario hizo clic en **Cancelar**, para lo cual se compara el resultado

con la constante static **CANCEL_OPTION**. Si son iguales, el programa termina. En la línea 84 se hace una llamada al método **getSelectedFile** de JfileChooser para recuperar un objeto File (paquete java.io) que representa el archivo o directorio que el usuario seleccionó, y luego se hace una llamada al método **toPath** de File para devolver un objeto Path. Después el programa muestra información acerca del archivo o directorio seleccionado.

```
1  // Fig. 15.12: DemoJFileChooser.java
2  // Demostración de la clase JFileChooser.
3  import java.io.IOException;
4  import java.nio.file.DirectoryStream;
5  import java.nio.file.Files;
6  import java.nio.file.Path;
7  import java.nio.file.Paths;
8  import javax.swing.JFileChooser;
9  import javax.swing.JFrame;
10 import javax.swing.JOptionPane;
11 import javax.swing.JScrollPane;
12 import javax.swing.JTextArea;
13
14 public class DemoJFileChooser extends JFrame
15 {
16    private final JTextArea areaSalida; // muestra el contenido del archivo
17
18    // establece la GUI
19    public DemoJFileChooser() throws IOException
20    {
21       super("Demo de JFileChooser");
22       areaSalida = new JTextArea();
23       add(new JScrollPane(areaSalida)); // areaSalida cuenta con controles
                                           deslizables
24       analizarRuta(); // obtiene el objeto Path del usuario y muestra la información
25    }
26
27    // muestra información acerca del archivo o directorio que especifica el usuario
28    public void analizarRuta() throws IOException
29    {
30       // crea un objeto Path con la ruta al archivo o directorio seleccionado
            por el usuario
31       Path ruta = obtenerRutaArchivoODirectorio();
32
33       if (ruta != null && Files.exists(ruta)) // si el nombre existe, muestra
                                                   información sobre él
34       {
35          // recopila información sobre el archivo (o directorio)
36          StringBuilder builder = new StringBuilder();
37          builder.append(String.format("%s:%n", path.getFileName()));
38          builder.append(String.format("%s un directorio%n",
39             Files.isDirectory(ruta) ? "Es" : "No es"));
40          builder.append(String.format("%s una ruta absoluta%n",
41             ruta.isAbsolute() ? "Es" : "No es"));
42          builder.append(String.format("Ultima modificacion: %s%n",
43             Files.getLastModifiedTime(ruta)));
44          builder.append(String.format("Tamanio: %s%n", Files.size(ruta)));
45          builder.append(String.format("Ruta: %s%n", ruta));
```

Fig. 15.12 | Demostración de JFileChooser (parte 1 de 2).

```
46            builder.append(String.format("Ruta absoluta: %s%n",
47               ruta.toAbsolutePath()));
48
49            if (Files.isDirectory(ruta)) // muestra en pantalla el listado del
                                           directorio
50            {
51               builder.append(String.format("%nContenido del directorio:%n"));
52
53               // objeto para iterar a través del contenido de un directorio
54               DirectoryStream<Path> flujoDirectorio =
55                  Files.newDirectoryStream(ruta);
56
57               for (Path p : flujoDirectorio)
58                  builder.append(String.format("%s%n", r));
59            }
60
61            areaSalida.setText(builder.toString()); // muestra el contenido del
                                                       objeto String
62         }
63         else // El objeto Path no existe
64         {
65            JOptionPane.showMessageDialog(this, ruta.getFileName() +
66               " no existe.", "ERROR", JOptionPane.ERROR_MESSAGE);
67         }
68      } // fin del método analizarRuta
69
70      // permite al usuario especificar el nombre del archivo o directorio
71      private Path obtenerRutaArchivoODirectorio()
72      {
73         // configura el diálogo para permitir la selección de un archivo o directorio
74         JFileChooser selectorArchivos = new JFileChooser();
75         selectorArchivos.setFileSelectionMode(
76            JFileChooser.FILES_AND_DIRECTORIES);
77         int resultado = selectorArchivos.showOpenDialog(this);
78
79         // si el usuario hizo clic en el botón Cancelar en el cuadro de diálogo,
           regresa
80         if (resultado == JFileChooser.CANCEL_OPTION)
81            System.exit(1);
82
83         // devuelve objeto Path que representa el archivo seleccionado
84         return selectorArchivo.getSelectedFile().toPath() ;
85      }
86   } // fin de la clase DemoJFileChooser
```

Fig. 15.12 | Demostración de `JFileChooser` (parte 2 de 2).

```
1  // Fig. 15.13: PruebaJFileChooser.java
2  // Prueba de la clase DemoJFileChooser.
3  import java.io.IOException;
4  import javax.swing.JFrame;
5
6  public class PruebaJFileChooser
7  {
```

Fig. 15.13 | Prueba de la clase `DemostracionFile` (parte 1 de 2).

```
8     public static void main(String[] args) throws IOException
9     {
10       DemoJFileChooser aplicacion = new DemoJFileChooser();
11       aplicacion.setSize(400, 400);
12       aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13       aplicacion.setVisible(true);
14    }
15 } // fin de la clase PruebaJFileChooser
```

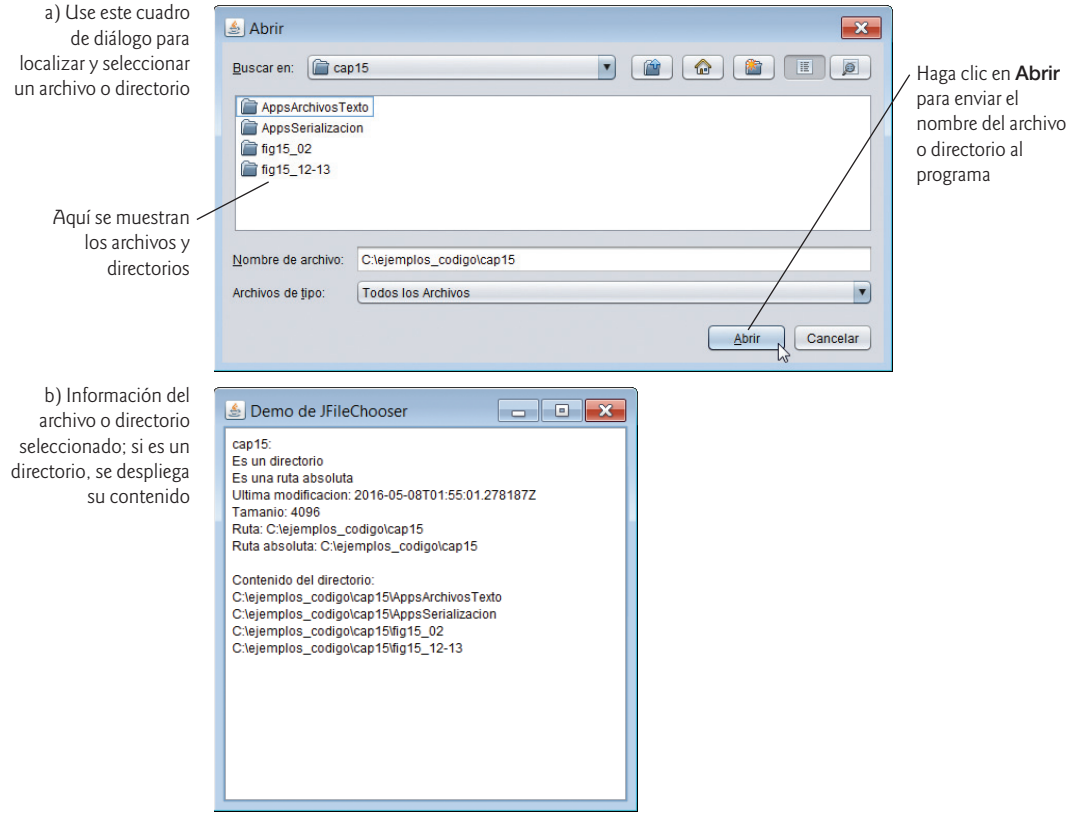

Fig. 15.13 | Prueba de la clase `DemostracionFile` (parte 2 de 2).

15.7 (Opcional) Clases adicionales de `java.io`

En esta sección veremos las generalidades de las interfaces y clases adicionales (del paquete `java.io`).

15.7.1 Interfaces y clases para entrada y salida basada en bytes

`InputStream` y `OutputStream` son clases `abstract` que declaran métodos para realizar operaciones basadas en bytes de entrada y salida, respectivamente.

Flujos de canalizaciones

Las **canalizaciones** son canales de comunicación sincronizados entre hilos. Hablaremos sobre los hilos en el capítulo 23 (en línea, en el sitio web del libro). Java proporciona las clases `PipedOutputStream` (una subclase

de `OutputStream`) y **`PipedInputStream`** (una subclase de `InputStream`) para establecer canalizaciones entre dos hilos en un programa. Un hilo envía datos a otro escribiendo a un objeto `PipedOutputStream`. El hilo de destino lee la información de la canalización mediante un objeto `PipedInputStream`.

Flujos de filtros

Un objeto **`FilterInputStream`** filtra a un objeto `InputStream`, y un objeto `FilterOutputStream` filtra a un objeto `OutputStream`. **Filtrar** significa simplemente que el flujo que actúa como filtro proporciona una funcionalidad adicional, como la incorporación de bytes de datos en unidades de tipo primitivo significativas. Por lo general, `FilterInputStream` y `FilterOutputStream` se usan como superclases, de manera que sus subclases proporcionan sus capacidades de filtrado.

Un objeto **`PrintStream`** (una subclase de `FilterOutputStream`) envía texto como salida hacia el flujo especificado. En realidad, hasta este punto hemos estado utilizando la salida mediante `PrintStream`, ya que `System.out` y `System.err` son objetos `PrintStream`.

Flujos de datos

Leer datos en forma de bytes sin ningún formato es un proceso rápido pero crudo. Por lo general los programas leen datos como agregados de bytes que forman valores `int`, `float`, `double` y así en lo sucesivo. Los programas de Java pueden utilizar varias clases para recibir datos de entrada y enviar datos de salida en forma de agregados.

La interfaz `DataInput` describe métodos para leer tipos primitivos desde un flujo de entrada. Las clases **`DataInputStream`** y `RandomAccessFile` implementan a esta interfaz para leer conjuntos de bytes y verlos como valores de tipo primitivo. La interfaz `DataInput` incluye métodos tales como `readBoolean`, `readByte`, `readChar`, `readDouble`, `readFloat`, `readFully` (para arreglos tipo `byte`), `readInt`, `readLong`, `readShort`, `readUnsignedByte`, `readUnsignedShort`, `readUTF` (para leer caracteres Unicode codificados por Java; hablaremos sobre la codificación UTF en el apéndice H), así como `skipBytes`.

La interfaz `DataOutput` describe un conjunto de métodos para escribir tipos primitivos hacia un flujo de salida. Las clases **`DataOutputStream`** (una subclase de `FilterOutputStream`) y `RandomAccessFile` implementan a esta interfaz para escribir valores de tipos primitivos como bytes. La interfaz `DataOutput` incluye versiones sobrecargadas del método `write` (para un `byte` o para un arreglo `byte`) y los métodos `writeBoolean`, `writeByte`, `writeBytes`, `writeChar`, `writeChars` (para objetos `String` Unicode), `writeDouble`, `writeFloat`, `writeInt`, `writeLong`, `writeShort` y `writeUTF` (para enviar texto modificado para Unicode).

Flujos con búfer

El **uso de un búfer** es una técnica para mejorar el rendimiento de las operaciones de E/S. Con un objeto **`BufferedOutputStream`** (una subclase de la clase `FilterOutputStream`), cada instrucción de salida *no* produce necesariamente una transferencia física real de datos hacia el dispositivo de salida (una operación lenta en comparación con las velocidades del procesador y de la memoria principal). En vez de ello cada operación de salida se dirige hacia una región en memoria conocida como **búfer**, que es lo bastante grande como para almacenar los datos de muchas operaciones de salida. Después, cada vez que se llena el búfer la transferencia real hacia el dispositivo de salida se realiza en una sola **operación física de salida** extensa. Las operaciones de salida dirigidas hacia el búfer de salida en memoria a menudo se conocen como **operaciones lógicas de salida**. Con un objeto `BufferedOutputStream`, se puede forzar a un búfer parcialmente lleno para que en cualquier momento envíe su contenido al dispositivo mediante la invocación del método **`flush`** del objeto flujo.

El uso de búfer puede aumentar de manera considerable la eficiencia de una aplicación. Las operaciones comunes de E/S son en extremo lentas si se les compara con la velocidad de acceso a los datos de la

memoria de la computadora. El uso de búfer reduce el número de operaciones de E/S, al combinar primero las operaciones de salida más pequeñas en la memoria. El número de operaciones físicas de E/S reales es pequeño en comparación con el número de solicitudes de E/S emitidas por el programa. Por ende, el programa que usa un búfer es más eficiente.

Tip de rendimiento 15.1

La E/S con búfer puede producir mejoras considerables en el rendimiento en comparación con la E/S sin búfer.

Con un objeto **`BufferedInputStream`** (una subclase de la clase `FilterInputStream`), muchos trozos "lógicos" de datos de un archivo se leen como una sola **operación física de entrada** extensa y se envían a un búfer de memoria. Conforme un programa solicita cada nuevo trozo de datos, éste se toma del búfer. (A este procedimiento se le conoce algunas veces como **operación lógica de entrada**). Cuando el búfer está vacío, se lleva a cabo la siguiente operación física de entrada real desde el dispositivo de entrada, con el fin de leer el siguiente grupo de trozos "lógicos" de datos. Por lo tanto el número de operaciones físicas de entrada reales es pequeño en comparación con el número de solicitudes de lectura emitidas por el programa.

Flujos de arreglos `byte` basados en memoria

La E/S de flujos en Java incluye herramientas para recibir datos de entrada de arreglos `byte` en memoria, y enviar datos de salida a arreglos `byte` en memoria. Un objeto `ByteArrayInputStream` (una subclase de `InputStream`) lee de un arreglo `byte` en memoria. Un objeto `ByteArrayOutputStream` (una subclase de `OutputStream`) escribe en un arreglo `byte` en memoria. Uno de los usos de la E/S con arreglos `byte` es la *validación de datos*. Un programa puede recibir como entrada una línea completa a la vez desde el flujo de entrada, para entonces colocarla en un arreglo `byte`. Después puede usarse una rutina de validación para analizar detalladamente el contenido del arreglo `byte` y corregir los datos, si es necesario. Por último, el programa puede recibir los datos de entrada del arreglo `byte` "sabiendo" que los datos de entrada se encuentran en el formato adecuado. Enviar datos de salida a un arreglo `byte` es una excelente manera de aprovechar las poderosas herramientas de formato para los datos de salida que proporcionan los flujos en Java. Por ejemplo, los datos pueden almacenarse en un arreglo `byte` utilizando el mismo formato que se mostrará posteriormente, y luego se puede enviar el arreglo `byte` hacia un archivo en disco para preservar la imagen en pantalla.

Secuenciamiento de la entrada desde varios flujos

Un objeto `SequenceInputStream` (una subclase de `InputStream`) permite la concatenación lógica de varios objetos `InputStream`. El programa ve a este grupo como un flujo `InputStream` continuo. Cuando el programa llega al final de un flujo de entrada, ese flujo se cierra y se abre el siguiente flujo en la secuencia.

15.7.2 Interfaces y clases para entrada y salida basada en caracteres

Además de los flujos basados en bytes, Java proporciona las clases `abstract` **`Reader`** y **`Writer`**, que son flujos basados en caracteres como los que utilizó para el procesamiento de archivos de texto en la sección 15.4. La mayoría de los flujos basados en bytes tienen sus correspondientes clases `Reader` o `Writer` concretas basadas en caracteres.

Objetos `Reader` y `Writer` con búfer basados en caracteres

Las clases **`BufferedReader`** (una subclase de la clase `abstract Reader`) y **`BufferedWriter`** (una subclase de la clase `abstract Writer`) permiten el uso del búfer para los flujos basados en caracteres. Recuerde que los flujos basados en caracteres utilizan caracteres Unicode, por lo que dichos flujos pueden procesar datos en cualquier lenguaje que sea representado por el conjunto de caracteres Unicode.

Objetos Reader y Writer para arreglos char basados en memoria

Las clases **CharArrayReader** y **CharArrayWriter** leen y escriben respectivamente un flujo de caracteres en un arreglo `char`. Un objeto **LineNumberReader** (una subclase de `BufferedReader`) es un flujo de caracteres con búfer que lleva el registro de los números de línea leídos; los caracteres de nueva línea, de retorno o las combinaciones de retorno de carro y avance de línea incrementan la cuenta de líneas. Si el programa necesita informar al lector sobre un error en una línea específica, puede ser útil llevar la cuenta de los números de línea.

Objetos Reader y Writer para archivos, canalizaciones y cadenas basadas en caracteres

Un objeto `InputStream` puede convertirse en un objeto `Reader` por medio de la clase **InputStreamReader**. De manera similar, un objeto `OutputStream` puede convertirse en un objeto `Writer` por medio de la clase **OutputStreamWriter**. Las clases `FileReader` (una subclase de `InputStreamReader`) y `FileWriter` (una subclase de `OutputStreamWriter`) leen y escriben caracteres en un archivo, respectivamente. Las clases **PipedReader** y **PipedWriter** implementan flujos de caracteres canalizados que pueden utilizarse para transferir datos entre hilos. Las clases **StringReader** y **StringWriter** leen y escriben caracteres, respectivamente, en objetos `String`. Un objeto `PrintWriter` escribe caracteres en un flujo.

15.8 Conclusión

En este capítulo aprendió a manipular datos persistentes. Comparamos los flujos basados en caracteres y los flujos basados en bytes, y presentamos varias clases de los paquetes `java.io` y `java.nio.file`. Utilizó las clases `Files` y `Paths` además de las interfaces `Path` y `DirectoryStream` para obtener información acerca de los archivos y directorios. Utilizó el procesamiento de archivos de acceso secuencial para manipular registros que se almacenan en orden, con base en el campo clave del registro. Conoció las diferencias entre el procesamiento de archivos de texto y la serialización de objetos, y utilizó la serialización para almacenar y obtener objetos completos. El capítulo concluyó con un pequeño ejemplo acerca del uso de un cuadro de diálogo `JFileChooser` para permitir a los usuarios seleccionar fácilmente archivos de una GUI. En el siguiente capítulo veremos las clases de Java para manipular colecciones de datos, como la clase `ArrayList` que presentamos en la sección 7.16.

Resumen

Sección 15.1 Introducción

- Las computadoras utilizan archivos para la retención a largo plazo de grandes cantidades de datos persistentes (pág. 645), incluso después de que los programas que crearon los datos terminan de ejecutarse.
- Las computadoras almacenan los archivos en dispositivos de almacenamiento secundario (pág. 645), como los discos duros.

Sección 15.2 Archivos y flujos

- Java ve a cada archivo como un flujo secuencial de bytes (pág. 645).
- Cada sistema operativo cuenta con un mecanismo para determinar el fin de un archivo, como un marcador de fin de archivo (pág. 645) o la cuenta de los bytes totales en el archivo.
- Los flujos basados en bytes (pág. 646) representan datos en formato binario.
- Los flujos basados en caracteres (pág. 646) representan datos como secuencias de caracteres.
- Los archivos que se crean usando flujos basados en bytes son archivos binarios (pág. 646). Los archivos que se crean usando flujos basados en caracteres son archivos de texto (pág. 646). Los archivos de texto se pueden leer mediante editores de texto, mientras que los archivos binarios se leen mediante un programa que convierte esos datos en un formato legible para el ser humano.

- Java también puede asociar los flujos con distintos dispositivos. Cuando un programa de Java empieza a ejecutarse, tres objetos flujo se asocian con dispositivos: `System.in`, `System.out` y `System.err`.

Sección 15.3 Uso de clases e interfaces NIO para obtener información de archivos y directorios

- Un objeto `Path` (pág. 647) representa la ubicación de un archivo o directorio. Los objetos de ruta no abren archivos ni proporcionan herramientas para procesamiento de archivos.
- La clase `Paths` (pág. 647) se utiliza para obtener un objeto `Path` que representa la ubicación de un archivo o directorio.
- La clase `Files` (pág. 647) proporciona métodos `static` para manipulaciones comunes de archivos y directorios, incluyendo métodos para copiar archivos, crear y eliminar tanto archivos como directorios, obtener información sobre archivos y directorios, leer el contenido de archivos, obtener objetos que nos permitan manipular el contenido de archivos y directorios, y varios más.
- Un objeto `DirectoryStream` (pág. 647) permite a un programa iterar a través del contenido de un directorio.
- El método `static get` (pág. 647) de la clase `Paths` convierte un objeto `String` que representa la ubicación de un archivo o directorio en un objeto `Path`.
- La entrada y salida basada en caracteres puede realizarse con las clases `Scanner` y `Formatter`.
- La clase `Formatter` (pág. 647) permite mostrar datos con formato en la pantalla o enviarlos a un archivo, de una manera similar a `System.out.printf`.
- Una ruta absoluta (pág. 647) contiene todos los directorios, empezando con el directorio raíz (pág. 647), que conducen hacia un archivo o directorio específico. Cada archivo o directorio en una unidad de disco tiene el mismo directorio raíz en su ruta.
- Una ruta relativa (pág. 647) empieza desde el directorio en el que se empezó a ejecutar la aplicación.
- El método `static exists` (pág. 648) de `Files` recibe un objeto `Path` y determina si existe (ya sea como archivo o directorio) en el disco.
- El método `getFileName` de `Path` (pág. 648) obtiene el nombre `String` de un archivo o directorio sin información sobre la ubicación.
- El método `static isDirectory` de `Files` (pág. 648) recibe un objeto `Path` y devuelve un `boolean` para indicar si ese objeto `Path` representa un directorio en el disco.
- El método `isAbsolute` de `Path` (pág. 648) devuelve un `boolean` que indica si un objeto `Path` representa una ruta absoluta a un archivo o directorio.
- El método `static getLastModifiedTime` de `Files` (pág. 648) recibe un objeto `Path` y devuelve un objeto `FileTime` (paquete `java.nio.file.atribute`) para indicar la última vez que se modificó el archivo.
- El método `static size` de `Files` (pág. 648) recibe un objeto `Path` y devuelve un `long` para representar el número de bytes en el archivo o directorio. En el caso de los directorios, el valor devuelto es específico de la plataforma.
- El método `toString` de `Path` (pág. 648) devuelve una representación `String` del objeto `Path`.
- El método `toAbsolutePath` de `Path` (pág. 648) convierte en una ruta absoluta el objeto `Path` desde el cual es llamado.
- El método `static newDirectoryStream` de `Files` (pág. 648) devuelve un objeto `DirectoryStream<Path>` que contiene objetos `Path` para el contenido de un directorio.
- Un carácter separador (pág. 650) se utiliza para separar directorios y archivos en la ruta.

Sección 15.4 Archivos de texto de acceso secuencial

- Java no impone una estructura en un archivo. El programador debe estructurar los archivos para satisfacer los requerimientos de una aplicación.
- Para obtener datos de un archivo en forma secuencial, los programas comúnmente empiezan desde el principio del archivo y leen todos los datos en forma consecutiva hasta encontrar la información deseada.
- Los datos en muchos archivos secuenciales no se pueden modificar sin el riesgo de destruir otros datos en el archivo. Por lo general, los registros en un archivo de acceso secuencial se actualizan volviendo a escribir el archivo completo.

Sección 15.5 Serialización de objetos

- Java cuenta con un mecanismo llamado serialización de objetos (pág. 662), el cual permite escribir o leer objetos completos mediante un flujo.
- Un objeto serializado (pág. 662) se representa como una secuencia de bytes e incluye los datos del objeto, así como información acerca del tipo del objeto y los tipos de datos que almacena.
- Una vez que se escribe un objeto serializado en un archivo, se puede leer del archivo y deserializarse (pág. 662) para recrearlo en la memoria.
- Las clases `ObjectInputStream` (pág. 662) y `ObjectOutputStream` (pág. 662) permiten leer o escribir objetos completos desde y hacia un flujo (posiblemente un archivo).
- Sólo las clases que implementan a la interfaz `Serializable` (pág. 663) pueden serializarse y deserializarse.
- La interfaz `ObjectOutput` (pág. 662) contiene el método `writeObject` (pág. 663), el cual recibe un `Object` como argumento y escribe su información en un `OutputStream`. Una clase que implementa a esta interfaz, como `ObjectOutputStream`, debe asegurarse que el objeto `Object` sea `Serializable`.
- La interfaz `ObjectInput` (pág. 662) contiene el método `readObject` (pág. 663), el cual lee y devuelve una referencia a un objeto `Object` a partir de un `InputStream`. Una vez que se lee un objeto, su referencia puede convertirse al tipo real del objeto.

Sección 15.6 Apertura de archivos con `JFileChooser`

- La clase `JFileChooser` (pág. 670) se utiliza para mostrar un cuadro de diálogo que permite a los usuarios de un programa seleccionar fácilmente archivos o directorios mediante una GUI.

Sección 15.7 (Opcional) Clases adicionales de `java.io`

- `InputStream` y `OutputStream` son clases `abstract` para realizar operaciones de E/S basadas en bytes.
- Las canalizaciones (pág. 673) son canales de comunicación sincronizados entre hilos. Un hilo envía datos mediante un `PipedOutputStream` (pág. 673). El hilo de destino lee información de la canalización mediante un `PipedInputStream` (pág. 673).
- Un flujo de filtro (pág. 674) proporciona una funcionalidad adicional, como la incorporación de bytes de datos en unidades de tipo primitivo significativas. Por lo general `FilterInputStream` (pág. 674) y `FilterOutputStream` se extienden de manera que sus subclases concretas proporcionan algunas de sus capacidades de filtrado.
- Un objeto `PrintStream` (pág. 674) envía texto como salida. `System.out` y `System.err` son objetos `PrintStream`.
- La interfaz `DataInput` describe métodos para leer tipos primitivos desde un flujo de entrada. Las clases `DataInputStream` (pág. 674) y `RandomAccessFile` implementan a esta interfaz.
- La interfaz `DataOutput` describe métodos para escribir tipos primitivos hacia un flujo de salida. Las clases `DataOutputStream` (pág. 674) y `RandomAccessFile` implementan a esta interfaz.
- El uso de búfer es una técnica para mejorar el rendimiento de E/S. Reduce el número de operaciones de E/S al combinar salidas más pequeñas y juntarlas en la memoria. El número de operaciones físicas de E/S es mucho menor que el número de peticiones de E/S emitidas por el programa.
- Con un objeto `BufferedOutputStream` (pág. 674), cada operación de salida se dirige hacia un búfer (pág. 674) lo bastante grande como para contener los datos de muchas operaciones de salida. Cada vez que se llena el búfer la transferencia al dispositivo de salida se realiza en una sola operación de salida física extensa (pág. 674). Es posible forzar el envío de datos de un búfer parcialmente lleno hacia el dispositivo en cualquier momento, invocando al método `flush` del objeto flujo (pág. 674).
- Con un objeto `BufferedInputStream` (pág. 675) muchos trozos "lógicos" de datos de un archivo se leen como una sola operación de entrada física extensa (pág. 675) y se colocan en un búfer de memoria. A medida que un programa solicita datos, éstos se obtienen del búfer. Cuando el búfer está vacío, se lleva a cabo la siguiente operación de entrada física real.
- Un objeto `ByteArrayInputStream` lee de un arreglo `byte` en memoria. Un objeto `ByteArrayOutputStream` escribe en un arreglo `byte` en memoria.

- Un objeto `SequenceInputStream` concatena varios objetos `InputStream`. Cuando el programa llega al final de un flujo de entrada, ese flujo se cierra y se abre el siguiente flujo en la secuencia.
- Las clases `abstract Reader` (pág. 675) y `Writer` (pág. 675) son flujos Unicode basados en caracteres. La mayoría de los flujos basados en caracteres tienen sus correspondientes clases `Reader` o `Writer` concretas basadas en caracteres.
- Las clases `BufferedReader` (pág. 675) y `BufferedWriter` (pág. 675) usan búfer con los flujos basados en caracteres.
- Las clases `CharArrayReader` (pág. 676) y `CharArrayWriter` (pág. 676) manipulan los arreglos `char`.
- Un objeto `LineNumberReader` (pág. 676) es un flujo de caracteres con búfer que lleva el registro de los números de línea leídos.
- Las clases `FileReader` (pág. 676) y `FileWriter` (pág. 676) realizan operaciones de E/S de archivos basadas en caracteres.
- Las clases `PipedReader` (pág. 676) y `PipedWriter` (pág. 676) implementan flujos de caracteres canalizados, que pueden utilizarse para transferir datos entre hilos.
- Las clases `StringReader` (pág. 676) y `StringWriter` (pág. 676) leen y escriben caracteres en objetos `String` respectivamente. Un objeto `PrintWriter` (pág. 654) escribe caracteres en un flujo.

Ejercicios de autoevaluación

15.1 Determine cuál de las siguientes proposiciones es *verdadera* o *falsa*. En caso de ser *falsa*, explique por qué.

a) Usted debe crear en forma explícita los objetos flujo `System.in`, `System.out` y `System.err`.
b) Al leer datos de un archivo mediante la clase `Scanner`, si desea leer varias veces datos en el archivo, éste debe cerrarse y volver a abrirse para leer desde el principio del archivo.
c) El método `static exists` de la clase `Files` recibe un objeto `Path` y determina si existe (ya sea como archivo o directorio) en el disco.
d) Los archivos binarios pueden ser leídos por los seres humanos en un editor de texto.
e) Una ruta absoluta contiene todos los directorios (empezando con el directorio raíz) que conducen hacia un archivo o directorio específico.
f) La clase `Formatter` contiene el método `printf`, que permite imprimir datos con formato en la pantalla, o enviarlos a un archivo.

15.2 Complete las siguientes tareas; suponga que cada una se aplica al mismo programa:

a) Escriba una instrucción que abra el archivo "`antmaest.txt`" en modo de entrada; use la variable `Scanner` llamada `entAntMaestro`.
b) Escriba una instrucción que abra el archivo "`trans.txt`" en modo de entrada; use la variable `Scanner` llamada `entTransaccion`.
c) Escriba una instrucción para abrir el archivo "`nuevomaest.txt`" en modo de salida (y creación); use la variable `Formatter` llamada `salNuevoMaest`.
d) Escriba las instrucciones necesarias para leer un registro del archivo "`antmaest.txt`". Los datos leídos deben usarse para crear un objeto de la clase `Cuenta`; use la variable `Scanner` llamada `entAntMaest`. Suponga que la clase `Cuenta` es la misma que la de la figura 15.9.
e) Escriba las instrucciones necesarias para leer un registro del archivo "`trans.txt`". El registro es un objeto de la clase `RegistroTransaccion`; use la variable `Scanner` llamada `entTransaccion`. Suponga que la clase `RegistroTransaccion` contiene el método `establecerCuenta` (que recibe un `int`) para establecer el número de cuenta, y contiene el método `establecerMonto` (que recibe un `double`) para establecer el monto de la transacción.
f) Escriba una instrucción que escriba un registro en el archivo "`nuevomaest.txt`". El registro es un objeto de tipo `Cuenta`; use la variable `Formatter` llamada `salNuevoMaest`.

15.3 Complete las siguientes tareas, suponiendo que cada una se aplica al mismo programa:

a) Escriba una instrucción que abra el archivo "`antmaest.ser`" en modo de entrada; use la variable `ObjectInputStream` llamada `entAntMaest` para envolver un objeto `InputStream`.
b) Escriba una instrucción que abra el archivo "`trans.ser`" en modo de entrada; use la variable `ObjectInputStream` llamada `entTransaccion` para envolver un objeto `InputStream`.

c) Escriba una instrucción para abrir el archivo "nuevomaest.ser" en modo de salida (y creación); use la variable ObjectOutputStream llamada salNuevoMaest para envolver un objeto OutputStream.
d) Escriba una instrucción que lea un registro del archivo "antmaest.ser". El registro es un objeto de la clase Cuenta; use la variable ObjectInputStream llamada entAntMaestro. Suponga que la clase Cuenta es igual que la clase Cuenta de la figura 15.9.
e) Escriba una instrucción que lea un registro del archivo "trans.ser". El registro es un objeto de la clase RegistroTransaccion; use la variable ObjectInputStream llamada entTransaccion.
f) Escriba una instrucción que escriba un registro de tipo Cuenta en el archivo "nuevomaest.ser"; use la variable ObjectOutputStream llamada salNuevoMaest.

Respuestas a los ejercicios de autoevaluación

15.1 a) Falso. Estos tres flujos se crean para el programador cuando se empieza a ejecutar una aplicación de Java.
b) Verdadero.
c) Verdadero.
d) Falso. Los archivos de texto pueden ser leídos por los seres humanos en un editor de texto. Los archivos binarios podrían ser leídos por los humanos, pero sólo si los bytes en el archivo representan caracteres ASCII.
e) Verdadero.
f) Falso. La clase Formatter contiene el método format el cual permite imprimir datos con formato en la pantalla, o enviarlos a un archivo.

15.2 a)
```
Scanner entAntMaest = new Scanner(Paths.get("antmaest.txt"));
```
b)
```
Scanner entTransaccion = new Scanner(Paths.get("trans.txt"));
```
c)
```
Formatter salNuevoMaest = new Formatter("nuevomaest.txt");
```
d)
```
Cuenta cuenta = new Cuenta();
cuenta.establecerCuenta(entAntMaest.nextInt());
cuenta.establecerPrimerNombre(entAntMaest.next());
cuenta.establecerApellidoPaterno(entAntMaest.next());
cuenta.establecerSaldo(entAntMaest.nextDouble());
```
e)
```
RegistroTransaccion transaccion = new Transaction();
transaccion.establecerCuenta(entTransaccion.nextInt());
transaccion.establecerMonto(entTransaccion.nextDouble());
```
f)
```
salNuevMaest.format("%d %s %s &.2f%n",
  cuenta.obtenerCuenta(), cuenta.obtenerPrimerNombre(),
  cuenta.obtenerApellidoPaterno(), cuenta.obtenerSaldo());
```

15.3 a)
```
ObjectInputStream entAntMaest = new ObjectInputStream(
  Files.newInputStream(Paths.get("antmaest.ser")));
```
b)
```
ObjectInputStream entTransaccion = new ObjectInputStream(
  Files.newOutputStream(Paths.get("trans.ser")));
```
c)
```
ObjectOutputStream salNuevMaest = new ObjectOutputStream(
  Files.newOutputStream("nuevmaest.ser")));
```
d)
```
Cuenta = (Cuenta) entAntMaest.readObject();
```
e)
```
registroTransaccion = (RegistroTransaccion) entTransaccion.readObject();
```
f)
```
salNuevMaest.writeObject(nuevaCuenta);
```

Ejercicios

15.4 ***(Asociación de archivos)*** El ejercicio de autoevaluación 15.2 pide al lector que escriba una serie de instrucciones individuales. En realidad estas instrucciones forman el núcleo de un tipo importante de programa para procesar archivos: un programa para asociar archivos. En el procesamiento de datos comercial, es común tener varios archivos en cada sistema de aplicaciones. Por ejemplo, en un sistema de cuentas por cobrar hay por lo general un archivo maestro que contiene información detallada acerca de cada cliente, como su nombre, dirección, número telefónico, saldo deudor, límite de crédito, términos de descuento, acuerdos contractuales y posiblemente un historial condensado de compras recientes y pagos en efectivo.

A medida que ocurren las transacciones (es decir, a medida que se generan las ventas y llegan los pagos en el correo), la información sobre éstas se introduce en un archivo. Al final de cada periodo de negocios (un mes para algunas compañías, una semana para otras y un día en algunos casos), el archivo de transacciones (llamado "`trans.txt`") se aplica al archivo maestro (llamado "`antmaest.txt`") para actualizar el registro de compras y pagos de cada cuenta. Durante una actualización, el archivo maestro se rescribe como el archivo "`nuevomaest.txt`", el cual se utiliza al final del siguiente periodo de negocios para empezar de nuevo el proceso de actualización.

Los programas para asociar archivos deben tratar con ciertos problemas que no existen en programas de un solo archivo. Por ejemplo, no siempre ocurre una asociación. Si un cliente en el archivo maestro no ha realizado compras ni pagos en efectivo en el periodo actual de negocios, no aparecerá ningún registro para este cliente en el archivo de transacciones. De manera similar, un cliente que haya realizado compras o pagos en efectivo podría haberse mudado recientemente a esta comunidad y tal vez la compañía no haya tenido la oportunidad de crear un registro maestro para este cliente.

Escriba un programa completo para asociar archivos de cuentas por cobrar. Utilice el número de cuenta en cada archivo como la clave de registro para fines de asociar los archivos. Suponga que cada archivo es un archivo de texto secuencial con registros almacenados en orden ascendente, por número de cuenta.

a) Defina la clase `RegistroTransaccion`. Los objetos de esta clase contienen un número de cuenta y un monto para la transacción. Proporcione métodos para modificar y obtener estos valores.
b) Modifique la clase `Cuenta` de la figura 15.9 para incluir el método `combinar`, el cual recibe un objeto `RegistroTransaccion` y combina el saldo del objeto `Cuenta` con el valor del monto del objeto `RegistroTransaccion`.
c) Escriba un programa para crear datos de prueba para el programa. Use los datos de la cuenta de ejemplo de las figuras 15.14 y 15.15. Ejecute el programa para crear los archivos `trans.txt` y `antmaest.txt`, para que los utilice su programa de asociación de archivos.

Archivo maestro Número de cuenta	Nombre	Saldo
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Susy Green	-14.22

Fig. 15.14 | Datos de ejemplo para el archivo maestro.

Archivo de transacciones Número de cuenta	Monto de la transacción
100	27.14
300	62.11
400	100.56
900	82.17

Fig. 15.15 | Datos de ejemplo para el archivo de transacciones.

d) Cree la clase `AsociarArchivos` para llevar a cabo la funcionalidad de asociación de archivos. La clase debe contener métodos para leer `antmaest.txt` y `trans.txt`. Cuando ocurra una coincidencia (es decir, que aparezcan registros con el mismo número de cuenta en el archivo maestro y en el archivo de

transacciones), sume el monto en dólares del registro de transacciones al saldo actual en el registro maestro y escriba el registro "`nuevomaest.txt`". (Suponga que las compras se indican mediante montos positivos en el archivo de transacciones y los pagos mediante montos negativos). Cuando haya un registro maestro para una cuenta específica pero no haya un registro de transacciones correspondiente, simplemente escriba el registro maestro en "`nuevomaest.txt`". Cuando haya un registro de transacciones pero no un registro maestro correspondiente, imprima en un archivo de registro el mensaje "`Hay un registro de transacciones no asociado para ese numero de cliente ...`" (utilice el número de cuenta del registro de transacciones). El archivo de registro debe ser un archivo de texto llamado "`registro.txt`".

15.5 ***(Asociación de archivos con varias transacciones)*** Es posible (y muy común) tener varios registros de transacciones con la misma clave de registro. Por ejemplo, esta situación ocurre cuando un cliente hace varias compras y pagos en efectivo durante un periodo de negocios. Rescriba su programa para asociar archivos de cuentas por cobrar del ejercicio 15.4, para ofrecer la posibilidad de manejar varios registros de transacciones con la misma clave de registro. Modifique los datos de prueba de `CrearDatos.java` para incluir los registros de transacciones adicionales de la figura 15.16.

Número de cuenta	Monto en dólares
300	83.89
700	80.78
700	1.53

Fig. 15.16 | Registros de transacciones adicionales.

15.6 ***(Asociación de archivos con serialización de objetos)*** Vuelva a crear su solución para el ejercicio 15.5, usando la serialización de objetos. Use las instrucciones del ejercicio 15.3 como base para este programa. Tal vez deba crear aplicaciones para que lean los datos almacenados en los archivos `.ser`; para ello puede modificar el código de la sección 15.5.2.

15.7 ***(Generador de palabras de números telefónicos)*** Los teclados telefónicos estándar contienen los dígitos del cero al nueve. Cada uno de los números del dos al nueve tiene tres letras asociadas (figura 15.17). A muchas personas se les dificulta memorizar números telefónicos, por lo que utilizan la correspondencia entre los dígitos y las letras para desarrollar palabras de siete letras que corresponden a sus números telefónicos. Por ejemplo, una persona cuyo número telefónico sea 686-3767 podría utilizar la correspondencia indicada en la figura 15.17 para desarrollar la palabra de siete letras "NUMEROS". Cada palabra de siete letras corresponde exactamente a un número telefónico de siete dígitos. El restaurante que desea incrementar su negocio de comidas para llevar podría lograrlo utilizando el número 266-4327 (es decir, "COMIDAS").

Dígito	Letras	Dígito	Letras	Dígito	Letras
2	A B C	5	J K L	8	T U V
3	D E F	6	M N O	9	W X Y
4	G H I	7	P Q R S		

Fig. 15.17 | Dígitos y letras de los teclados telefónicos.

Cada número telefónico de siete letras corresponde a muchas palabras de siete letras distintas, pero la mayoría de estas palabras representan yuxtaposiciones irreconocibles de letras. Sin embargo, es posible que el dueño de una

carpintería se complazca en saber que el número telefónico de su taller, 683-2537, corresponde con "MUEBLES". El propietario de una tienda de licores estaría sin duda feliz de averiguar que el número telefónico 232-4327 corresponde a "BEBIDAS". Un veterinario con el número telefónico 627-2682 se complacería en saber que ese número corresponde a las letras "MASCOTA". El propietario de una tienda de música estaría complacido en saber que su número telefónico 687-4225 corresponde a "MUSICAL".

Escriba un programa que a partir de un número dado de siete dígitos utilice un objeto `PrintStream` para escribir en un archivo todas las combinaciones posibles de palabras de siete letras que corresponden a ese número. Hay 2,187 (3^7) combinaciones posibles. Evite los números telefónicos con los dígitos `0` y `1`.

15.8 ***(Encuesta estudiantil)*** La figura 7.8 contiene un arreglo de respuestas a una encuesta, el cual está codificado directamente en el programa. Suponga que deseamos procesar resultados de encuestas que se guarden en un archivo. Este ejercicio requiere de dos programas separados. Primero cree una aplicación que pida al usuario las respuestas de la encuesta y que escriba cada respuesta en un archivo. Utilice un objeto `Formatter` para crear un archivo llamado `numeros.txt`. Cada entero debe escribirse utilizando el método `format`. Después modifique el programa de la figura 7.8 para leer las respuestas de la encuesta del archivo `numeros.txt`. Las respuestas deben leerse del archivo mediante el uso de un objeto `Scanner`. Use el método `nextInt` para introducir un entero del archivo a la vez. El programa deberá seguir leyendo respuestas hasta que llegue al fin del archivo. Los resultados deberán escribirse en el archivo de texto "`salida.txt`".

15.9 ***(Agregar serialización de objetos a la aplicación de dibujo `MiFigura`)*** Modifique el ejercicio 12.17 para permitir que el usuario guarde un dibujo en un archivo, o cargue un dibujo anterior de un archivo, usando la serialización de objetos. Agregue los botones `Cargar` (para leer objetos de un archivo) y `Guardar` (para escribir objetos en un archivo). Use un objeto `ObjectOutputStream` para escribir en el archivo y un objeto `ObjectInputStream` para leer del archivo. Escriba el arreglo de objetos `MiFigura` usando el método `writeObject` (clase `ObjectOutputStream`) y lea el arreglo usando el método `readObject` (`ObjectInputStream`). El mecanismo de serialización de objetos puede leer o escribir arreglos completos; no es necesario manipular cada elemento del arreglo de objetos `MiFigura` por separado. Simplemente se requiere que todas las figuras sean `Serializable`. Para los botones `Cargar` y `Guardar`, use un objeto `JfileChooser` para permitir que el usuario seleccione el archivo en el que se almacenarán las figuras, o del que se leerán. Cuando el usuario ejecute el programa por primera vez, no se mostrarán figuras en la pantalla. El usuario puede mostrar figuras abriendo un archivo de figuras previamente guardado, o puede dibujar nuevas figuras. Una vez que haya figuras en la pantalla, los usuarios podrán guardarlas en un archivo, usando el botón `Guardar`.

Marcando la diferencia

15.10 ***(Explorador de phishing)*** El *phishing* es una forma de robo de identidad, en la que mediante un correo electrónico el emisor se hace pasar por una fuente confiable e intenta adquirir información privada como nombres de usuario, contraseñas, números de tarjetas de crédito y número de seguro social. Los correos electrónicos de phishing que afirman provenir de bancos, compañías de tarjetas de crédito, sitios de subastas, redes sociales y sistemas de pago en línea populares pueden tener una apariencia bastante legítima. A menudo, estos mensajes fraudulentos proveen vínculos a sitios Web falsificados, en donde se pide al usuario que introduzca información confidencial.

Investigue estafas de phishing en línea. Visite también el sitio Web del Grupo de trabajo anti-phishing (`www.antiphishing.org/`) y el sitio Web de Investigaciones cibernéticas del FBI (`www.fbi.gov/cyberinvest/cyberhome.htm`), en donde encontrará información sobre las estafas más recientes y cómo protegerse a sí mismo.

Cree una lista de 30 palabras, frases y nombres de compañías que se encuentren frecuentemente en los mensajes de phishing. Asigne el valor de un punto a cada una, con base en una estimación de la probabilidad de que aparezca en un mensaje de phishing (por ejemplo, un punto si es poco probable, dos puntos si es algo probable, o tres puntos si es muy probable). Escriba una aplicación que explore un archivo de texto en busca de estos términos y frases. Para cada ocurrencia de una palabra clave o frase dentro del archivo de texto, agregue el valor del punto asignado al total de puntos para esa palabra o frase. Para cada palabra clave o frase encontrada, imprima en pantalla una línea con la palabra o frase, el número de ocurrencias y el total de puntos. Después muestre el total de puntos para todo el mensaje. ¿Asigna su programa un total de puntos alto a ciertos mensajes reales de correo electrónico de phishing que haya recibido? ¿Asigna un total de puntos alto a ciertos correos electrónicos legítimos que haya recibido?