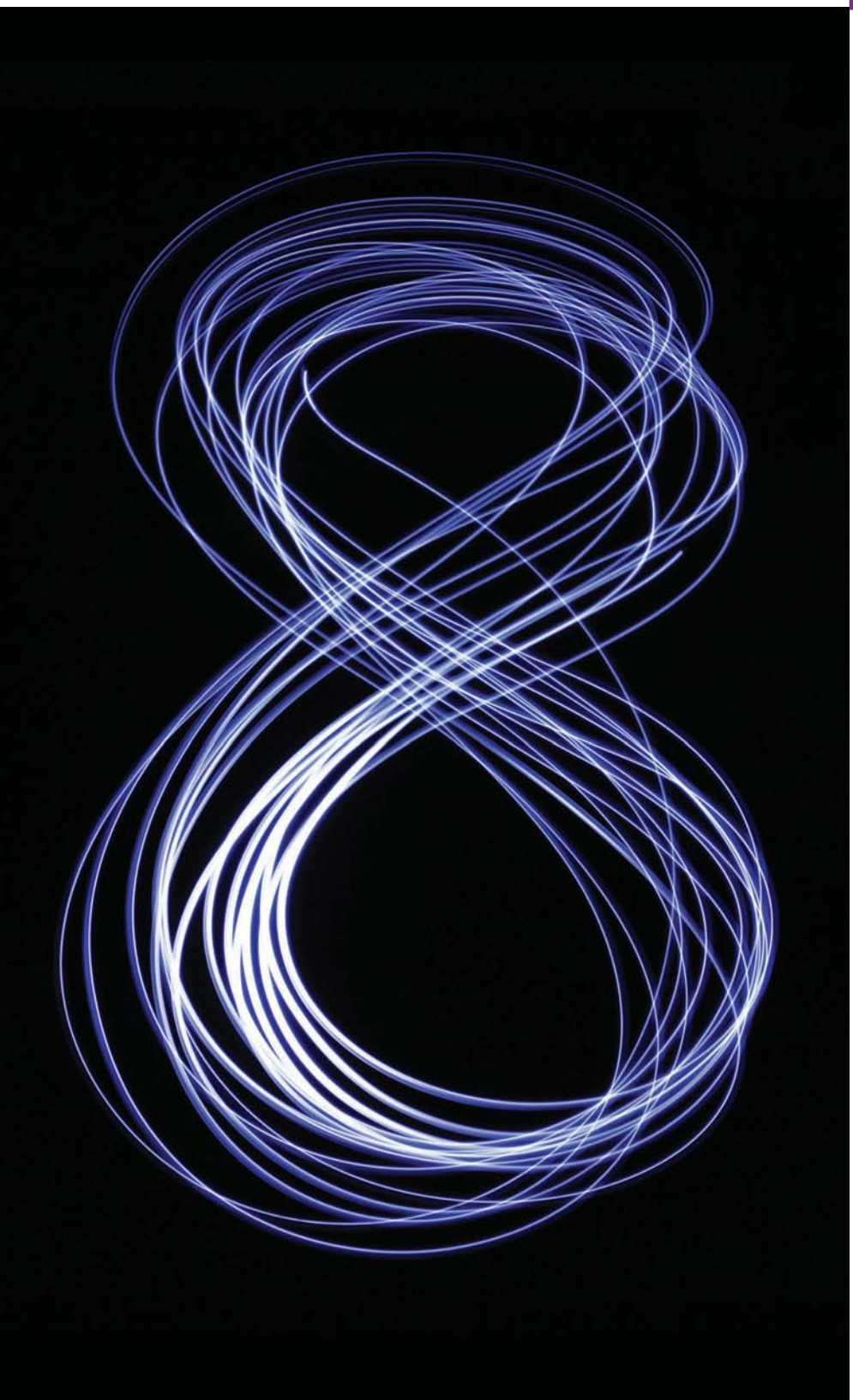
Arquivos, fluxos e serialização de objetos

15



A consciência … em si não aparece dividida em pedaços [bits]. Um "rio" ou um "fluxo" são as metáforas com que ela é mais naturalmente descrita.

— William James

Objetivos

Neste capítulo, você irá:

- Criar, ler, gravar e atualizar arquivos.
- Recuperar informações sobre arquivos e diretórios usando os recursos das NIO.2 APIs.
- Aprender as diferenças entre arquivos de texto e arquivos binários.
- Usar a classe `Formatter` a fim de gerar texto para um arquivo.
- Utilizar a classe `Scanner` para inserir texto de um arquivo.
- Escrever e ler objetos a partir de um arquivo usando serialização de objeto, a interface `Serializable` e as classes `ObjectOutputStream` e `ObjectInputStream`.
- Usar um diálogo `JFileChooser` para permitir que os usuários selecionem arquivos ou diretórios no disco.

Sumário

15.1 Introdução

Dados armazenados em variáveis e arrays são *temporários* — eles são perdidos quando uma variável local "sai do escopo" ou quando o programa termina. Para retenção de longo prazo dos dados, mesmo depois de os programas que os criaram serem fechados, os computadores usam **arquivos**. Você utiliza arquivos diariamente para tarefas como escrever um documento ou criar uma planilha. Computadores armazenam arquivos em **dispositivos de armazenamento secundário**, incluindo discos rígidos, pen-drives, DVDs etc. Os dados mantidos nos arquivos são **dados persistentes** — eles continuam existindo depois da execução do programa. Neste capítulo, explicaremos como programas Java criam, atualizam e processam arquivos.

Começaremos com uma discussão a respeito da arquitetura do Java para lidar com arquivos de maneira programática. Então, explicaremos que os dados podem ser armazenados em *arquivos de texto* e *arquivos binários* — e abrangeremos as diferenças entre eles. Demonstraremos como recuperar informações sobre arquivos e diretórios usando classes `Paths` e `Files` e interfaces `Path` e `DirectoryStream` (todas do pacote `java.nio.file`), depois vamos considerar os mecanismos para gravar e ler dados de arquivos. Mostraremos como criar e manipular arquivos de texto de acesso sequencial. Trabalhar com eles permite que você comece a manipular arquivos rápida e facilmente. Como você aprenderá, porém, é difícil ler dados a partir de arquivos de texto de volta na forma de objetos. Felizmente, várias linguagens orientadas a objetos (incluindo Java) fornecem maneiras de gravá-los e lê-los de arquivos (conhecidas como *serialização* e *desserialização de objetos*). Para ilustrar isso, recriamos alguns de nossos programas de acesso sequencial que utilizaram arquivos de texto, dessa vez armazenando e recuperando objetos de arquivos binários.

15.2 Arquivos e fluxos

O Java vê cada arquivo como um **fluxo de bytes** sequencial (Figura 15.1).[1] Cada sistema operacional fornece um mecanismo para determinar o fim de um arquivo, como um **marcador de fim de arquivo** ou uma contagem dos bytes totais no arquivo que é registrado em uma estrutura de dados administrativa mantida pelo sistema. Um programa Java que processa um fluxo de bytes simplesmente recebe uma indicação do sistema operacional quando ele alcança o fim desse fluxo — o programa *não* precisa saber como a plataforma subjacente representa arquivos ou fluxos. Em alguns casos, a indicação de fim de arquivo ocorre como uma exceção. Em outros, a indicação é um valor de retorno de um método invocado sobre um objeto que processa o fluxo.

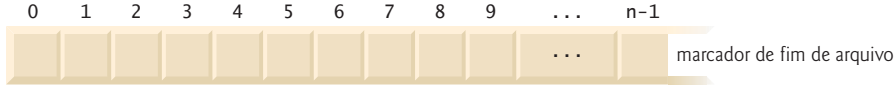


Figura 15.1 | Visualização do Java de um arquivo de *n* bytes.

[1] As APIs NIO do Java incluem também classes e interfaces que implementam a chamada arquitetura baseada em canal para entrada e saída de alto desempenho. Esses temas estão além do escopo deste livro.

Fluxos baseados em caracteres e em bytes

Fluxos de arquivos podem ser utilizados para entrada e saída de dados como bytes ou caracteres.

- **Fluxos baseados em bytes** geram e inserem dados em um formato *binário* — um `char` tem dois bytes, um `int` tem quatro bytes, um `double` tem oito bytes etc.
- **Fluxos baseados em caracteres** geram e inserem dados como uma *sequência de caracteres* na qual cada caractere tem dois bytes — o número de bytes para determinado valor depende do número de caracteres nesse valor. Por exemplo, o valor `2000000000` requer 20 bytes (10 caracteres a dois bytes por caractere), mas o valor `7` só demanda dois bytes (um caractere a dois bytes por caractere).

Arquivos criados com base nos fluxos de bytes são chamados **arquivos binários**, e aqueles criados com base nos fluxos de caracteres são **arquivos de texto**. Estes últimos podem ser lidos por editores de texto, enquanto os primeiros, por programas que entendem o conteúdo específico do arquivo e seu ordenamento. Um valor numérico em um arquivo binário pode ser usado em cálculos, ao passo que o caractere `5` é simplesmente um caractere que pode ser utilizado em uma string de texto, como em `"Sarah Miller is 15 years old"`.

Fluxos de entrada, saída e erro padrão

Um programa Java **abre** um arquivo criando e associando um objeto ao fluxo de bytes ou de caracteres. O construtor do objeto interage com o sistema operacional para *abrir* esse arquivo. O Java também pode associar fluxos a diferentes dispositivos. Quando um programa Java começa a executar, ele cria três objetos de fluxo que estão relacionados com dispositivos — `System.in`, `System.out` e `System.err`. O objeto `System.in` (fluxo de entrada padrão) normalmente permite que um programa insira bytes a partir do teclado. Já o objeto `System.out` (o objeto de fluxo de saída padrão), em geral, possibilita que um programa envie caracteres para a tela. Por fim, o objeto `System.err` (o objeto de fluxo de erro padrão) na maioria das vezes autoriza que um programa gere mensagens de erro baseadas em caracteres e as envie para a tela. Cada fluxo pode ser **redirecionado**. Para `System.in`, essa capacidade libera o programa a fim de ler bytes a partir de uma origem diferente. Para `System.out` e `System.err`, ele permite que a saída seja enviada a um local diferente, como a um arquivo em disco. A classe `System` fornece os métodos **`setIn`**, **`setOut`** e **`setErr`** para redirecionar os fluxos de entrada, saída e erro padrão, respectivamente.

Pacotes `java.io` e `java.nio`

Programas Java executam o processamento baseado em fluxo com classes e interfaces do pacote **`java.io`** e subpacotes do **`java.nio`** — novas APIs de E/S do Java introduzidas pela primeira vez no Java SE 6 e que desde então foram aprimoradas. Há também outros pacotes por todas as APIs Java que contêm classes e interfaces baseadas naquelas dos pacotes `java.io` e `java.nio`.

A entrada e saída baseada em caracteres pode ser realizada com as classes `Scanner` e **`Formatter`**, como veremos na Seção 15.4. Você usou a classe `Scanner` extensivamente para inserir dados a partir do teclado. `Scanner` também pode ler dados de um arquivo. A classe `Formatter` permite que a saída de dados formatados seja enviada para qualquer fluxo baseado em texto de uma maneira semelhante ao método `System.out.printf`. O Apêndice I (em inglês, na Sala Virtual do livro) apresenta os detalhes da saída formatada com `printf`. Todos esses recursos também podem ser utilizados para formatar arquivos de texto. No Capítulo 28 (em inglês, na Sala Virtual do livro), utilizaremos classes de fluxo para implementar aplicativos de rede.

Java SE 8 adiciona outro tipo de fluxo

O Capítulo 17, "Lambdas e fluxos do Java SE 8", introduz um novo tipo de fluxo que é usado para processar coleções de elementos (como arrays e `ArrayLists`), em vez dos fluxos de bytes discutidos nos exemplos de processamento de arquivo deste capítulo.

15.3 Usando classes e interfaces NIO para obter informações de arquivo e diretório

As interfaces `Path` e `DirectoryStream` e as classes `Paths` e `Files` (todas do pacote `java.nio.file`) são úteis para recuperar informações sobre arquivos e diretórios no disco:

- Interface **`Path`** — os objetos das classes que implementam essa interface representam o local de um arquivo ou diretório. Objetos `Path` não abrem arquivos nem fornecem capacidades de processamento deles.
- Classe **`Paths`** — fornece os métodos `static` utilizados para obter um objeto `Path` representando um local de arquivo ou diretório.
- Classe **`Files`** — oferece os métodos `static` para manipulações de arquivos e diretórios comuns, como copiar arquivos; criar e excluir arquivos e diretórios; obter informações sobre arquivos e diretórios; ler o conteúdo dos arquivos; obter objetos que permitem manipular o conteúdo de arquivos e diretórios; e mais.
- Interface **`DirectoryStream`** — os objetos das classes que implementam essa interface possibilitam que um programa itere pelo conteúdo de um diretório.

Criando objetos `Path`

Você usará a classe `static` do método **`get`** da classe `Paths` para converter uma `String` que representa o local de um arquivo ou diretório em um objeto `Path`. Você pode então usar os métodos da interface `Path` e classe `Files` para determinar informações sobre o arquivo ou diretório especificado. Discutiremos vários desses métodos mais adiante. Para listas completas dos 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
```

Caminhos absolutos versus *relativos*

Um caminho de arquivo ou diretório especifica sua localização em disco. Ele inclui alguns ou todos os principais diretórios que levam ao arquivo ou diretório. Um **caminho absoluto** contém *todos* os diretórios, desde o **diretório raiz**, que encaminha a um arquivo ou diretório específico. Cada arquivo ou diretório em uma unidade de disco particular tem o *mesmo* diretório-raiz em seu caminho. Um **caminho relativo** é "relativo" a outro diretório; por exemplo, um caminho relativo para o diretório em que o aplicativo começou a executar.

Obtendo objetos `Path` *de URIs*

Uma versão sobrecarregada do método `static` `Files` usa um objeto `URI` para localizar o arquivo ou diretório. Um **Uniform Resource Identifier (URI)** é uma forma mais geral dos **Uniform Resource Locators (URLs)** que são utilizados para pesquisar sites na web. Por exemplo, o URL `<http://www.deitel.com/>` direciona para o site da Deitel & Associates. Os URIs para encontrar arquivos variam em diferentes sistemas operacionais. Em plataformas Windows, o URI

```
file://C:/data.txt
```

identifica o arquivo `data.txt` armazenado no diretório-raiz da unidade C:. Em plataformas UNIX/Linux, o URI

```
file:/home/student/data.txt
```

identifica o arquivo `data.txt` armazenado no diretório `home` do usuário `student`.

Exemplo: obtendo informações de arquivo e diretório

A Figura 15.2 solicita que o usuário insira um nome de arquivo ou diretório, então usa as classes `Paths`, `Path`, `Files` e `DirectoryStream` para produzir informações sobre esse arquivo ou diretório. O programa começa solicitando ao usuário um arquivo ou diretório (linha 16). A linha 19 insere o nome de arquivo ou diretório e o passa para o método `Paths` `static` `get`, que converte a `String` em um `Path`. A linha 21 invoca o método `Files` `static` **`exists`**, que recebe um `Path` e determina se ele existe (como um arquivo ou como um diretório) no disco. Se o nome não existir, o controle passa para a linha 49, que exibe uma mensagem contendo a representação `String` de `Path` seguida por "`does not exist`". Caso contrário, as linhas 24 a 45 executam:

- O método `Path` **`getFileName`** (linha 24), que recebe o nome `String` do arquivo ou diretório sem nenhuma informação sobre o local.
- O método `Files` `static` **`isDirectory`** (linha 26), que recebe um `Path` e retorna um `boolean` indicando se esse `Path` representa um diretório no disco.
- O método `Path` **`isAbsolute`** (linha 28), que retorna um `boolean` indicando se esse `Path` representa um caminho absoluto para um arquivo ou diretório.
- O método `Files` `static` **`getLastModifiedTime`** (linha 30), que recebe um `Path` e retorna um `FileTime` (pacote `java.nio.file.attribute`), indicando quando o arquivo foi modificado pela última vez. O programa gera uma saída da representação `String` padrão de `FileTime`.
- O método `Files` `static` **`size`** (linha 31), que recebe um `Path` e retorna um `long` representando o número de bytes no arquivo ou diretório. Para diretórios, o valor retornado é específico da plataforma.
- O método `Path` **`toString`** (chamado implicitamente na linha 32), que retorna uma `String` representando o `Path`.
- O método `Path` **`toAbsolutePath`** (linha 33), que converte o `Path` em que ele é chamado para um caminho absoluto.

Se o `Path` representa um diretório (linha 35), as linhas 40 e 41 usam o método `Files` `static` **`newDirectoryStream`** (linhas 40 e 41) para obter um `DirectoryStream<Path>` contendo os objetos `Path` ao conteúdo do diretório. As linhas 43 e 44 exibem a representação `String` de cada `Path` em `DirectoryStream<Path>`. Observe que `DirectoryStream` é um tipo genérico como `ArrayList` (Seção 7.16).

A primeira saída desse programa demonstra um `Path` para a pasta que contém os exemplos deste capítulo. Já a segunda saída aponta um `Path` para o arquivo de código-fonte desse exemplo. Nos dois casos, especificamos um caminho absoluto.

```java
// Figura 15.2: FileAndDirectoryInfo.java
// A classe File utilizada para obter informações de arquivo e de diretório.
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;

public class FileAndDirectoryInfo
{
   public static void main(String[] args) throws IOException
   {
      Scanner input = new Scanner(System.in);

      System.out.println("Enter file or directory name:");

      // cria o objeto Path com base na entrada de usuário
      Path path = Paths.get(input.nextLine());

      if (Files.exists(path)) // se o caminho existe, gera uma saída das informações sobre ele
      {
         // exibe informações sobre o arquivo (ou diretório)
         System.out.printf("%n%s exists%n", path.getFileName());
         System.out.printf("%s a directory%n",
            Files.isDirectory(path) ? "Is" : "Is not");
         System.out.printf("%s an absolute path%n",
            path.isAbsolute() ? "Is" : "Is not");
         System.out.printf("Last modified: %s%n",
            Files.getLastModifiedTime(path));
         System.out.printf("Size: %s%n", Files.size(path));
         System.out.printf("Path: %s%n", path);
         System.out.printf("Absolute path: %s%n", path.toAbsolutePath());

         if (Files.isDirectory(path)) // listagem de diretório de saída
         {
            System.out.printf("%nDirectory contents:%n");

            // objeto para iteração pelo conteúdo de um diretório
            DirectoryStream<Path> directoryStream =
               Files.newDirectoryStream(path);

            for (Path p : directoryStream)
               System.out.println(p);
         }
      }
      else // se não for arquivo ou diretório, gera saída da mensagem de erro
      {
         System.out.printf("%s does not exist%n", path);
      }
   } // fim de main
} // fim da classe FileAndDirectoryInfo
```

```
Enter file or directory name:
c:\examples\ch15

ch15 exists
Is a directory
Is an absolute path
Last modified: 2013-11-08T19:50:00.838256Z
Size: 4096
Path: c:\examples\ch15
Absolute path: c:\examples\ch15

Directory contents:
C:\examples\ch15\fig15_02
C:\examples\ch15\fig15_12_13
C:\examples\ch15\SerializationApps
C:\examples\ch15\TextFileApps
```

continua

continuação

```
Enter file or directory name:
C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java

FileAndDirectoryInfo.java exists
Is not a directory
Is an absolute path
Last modified: 2013-11-08T19:59:01.848255Z
Size: 2952
Path: C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java
Absolute path: C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java
```

Figura 15.2 | A classe `File` utilizada para obter informações de arquivo e de diretório.

Dica de prevenção de erro 15.1

Depois de confirmar que um `Path` existe, ainda é possível que os métodos demonstrados na Figura 15.2 lancem `IOException`s. Por exemplo, o arquivo ou diretório representado pelo `Path` pode ser excluído do sistema após a chamada para o método `Files` `exists` e antes que as outras instruções nas linhas 24 a 45 sejam executadas. Programas de produção robustos com processamento de arquivos e diretórios exigem tratamento de exceção extenso para se recuperarem dessas possibilidades.

Caracteres separadores

Um **caractere separador** é utilizado para separar diretórios e arquivos em um caminho. Em um computador Windows, o *caractere separador* é uma barra invertida (\). Em um sistema Linux ou Mac OS X, é uma barra (/). O Java processa esses dois caracteres de maneira idêntica em um nome de caminho. Por exemplo, se fôssemos utilizar o caminho

```
c:\Program Files\Java\jdk1.6.0_11\demo/jfc
```

que emprega cada caractere separador, ainda assim o Java processaria o caminho adequadamente.

Boa prática de programação 15.1

Ao criar `String`s que representam informações de caminho, utilize `File.separator` para obter o caractere de separador adequado do computador local, em vez de utilizar explicitamente `/` ou `\`. Essa constante é uma `String` que consiste em um caractere — o separador apropriado para o sistema.

Erro comum de programação 15.1

Utilizar `\` como um separador de diretório em vez de `\\` em uma literal de string é um erro de lógica. Uma `\` simples indica que a `\` seguida pelo próximo caractere representa uma sequência de escape. Utilize `\\` para inserir um `\` em uma literal de string.

15.4 Arquivos de texto de acesso sequencial

A seguir, vamos criar e manipular *arquivos de acesso sequencial* em que os registros são armazenados na ordem pelo campo chave de registro. Começamos com *arquivos de texto,* permitindo que o leitor crie e edite arquivos rapidamente legíveis. Discutimos como criar, gravar e ler dados, além de atualizar arquivos de texto de acesso sequencial. Também incluímos um programa de consulta de crédito que recupera dados de um arquivo. Todos os programas nas seções 15.4.1 a 15.4.3 estão no diretório `TextFileApps` do capítulo de modo que possam manipular o mesmo arquivo de texto, que também está armazenado no diretório.

15.4.1 Criando um arquivo de texto de acesso sequencial

O Java não impõe nenhuma estrutura a um arquivo — noções como registros não fazem parte da linguagem Java. Portanto, você deve estruturar os arquivos para atender os requisitos dos seus aplicativos. No exemplo a seguir, veremos como impor uma estrutura de registro *chaveado* a um arquivo.

O programa desta seção cria um arquivo de acesso sequencial simples que pode ser usado em um sistema de contas a receber para monitorar os valores devidos a uma empresa por seus clientes creditícios. Para cada cliente, o programa obtém do usuário um número de conta, o nome e o saldo do cliente (isto é, o valor que o cliente deve à empresa por bens e serviços recebidos). Os dados de cada cliente constituem um "registro" para ele. Esse aplicativo utiliza o número de conta como a *chave de registro* — os registros do arquivo serão criados e mantidos na ordem dos números das contas. O programa assume que o usuário insere os registros

em ordem de número de conta. Em um sistema abrangente de contas a receber (baseado em arquivos de acesso sequencial), seria fornecido um recurso de *classificação*, de modo que o usuário pudesse inserir o registro em *qualquer* ordem. Os registros seriam, então, classificados e gravados no arquivo.

Classe `CreateTextFile`

A classe `CreateTextFile` (Figura 15.3) usa um `Formatter` para gerar `String`s formatadas utilizando as mesmas capacidades de formatação que as do método `System.out.printf`. Um objeto `Formatter` pode gerar saída para vários locais, como para uma janela de comando ou um arquivo, como fazemos neste exemplo. O objeto `Formatter` é instanciado na linha 26 no método `openFile` (linhas 22 a 38). O construtor utilizado na linha 26 recebe um argumento — uma `String` contendo o nome do arquivo, incluindo seu caminho. Se um caminho não for especificado, como é o caso aqui, a JVM assume que o arquivo está no diretório a partir do qual o programa foi executado. Para arquivos de texto, utilizamos a extensão de arquivo `.txt`. Se o arquivo *não* existir, ele será *criado*. Se um arquivo *existente* estiver aberto, seu conteúdo será **truncado** — todos os dados no arquivo são *descartados*. Se nenhuma exceção ocorrer, o arquivo é aberto para gravação e o objeto `Formatter` resultante pode ser usado a fim de gravar dados no arquivo.

```
1  // Figura 15.3: CreateTextFile.java
2  // Gravando dados em um arquivo de texto sequencial com a classe 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 CreateTextFile
11 {
12    private static Formatter output; // envia uma saída de texto para um arquivo
13
14    public static void main(String[] args)
15    {
16       openFile();
17       addRecords();
18       closeFile();
19    }
20
21    // abre o arquivo clients.txt
22    public static void openFile()
23    {
24       try
25       {
26          output = new Formatter("clients.txt"); // abre o arquivo
27       }
28       catch (SecurityException securityException)
29       {
30          System.err.println("Write permission denied. Terminating.");
31          System.exit(1); // termina o programa
32       }
33       catch (FileNotFoundException fileNotFoundException)
34       {
35          System.err.println("Error opening file. Terminating.");
36          System.exit(1); // termina o programa
37       }
38    }
39
40    // adiciona registros ao arquivo
41    public static void addRecords()
42    {
43       Scanner input = new Scanner(System.in);
44       System.out.printf("%s%n%s%n? ",
45          "Enter account number, first name, last name and balance.",
46          "Enter end-of-file indicator to end input.");
47
48       while (input.hasNext()) // faz um loop até o indicador de fim de arquivo
49       {
```

continua

continuação

```
50            try
51            {
52               // gera saída do novo registro para o arquivo; supõe entrada válida
53               output.format("%d %s %s %.2f%n", input.nextInt(),
54                  input.next(), input.next(), input.nextDouble());
55            }
56            catch (FormatterClosedException formatterClosedException)
57            {
58               System.err.println("Error writing to file. Terminating.");
59               break;
60            }
61            catch (NoSuchElementException elementException)
62            {
63               System.err.println("Invalid input. Please try again.");
64               input.nextLine(); // descarta entrada para o usuário tentar de novo
65            }
66
67            System.out.print("? ");
68         } // fim do while
69      } // fim do método addRecords
70
71      // fecha o arquivo
72      public static void closeFile()
73      {
74         if (output != null)
75            output.close();
76      }
77   } // fim da classe CreateTextFile
```

```
Enter account number, first name, last name and balance.
Enter end-of-file indicator to end input.
? 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
```

Figura 15.3 | Gravando dados em um arquivo de texto sequencial com a classe `Formatter`.

As linhas 28 a 32 tratam da `SecurityException`, que ocorre se o usuário não tiver permissão para gravar dados no arquivo. As linhas 33 a 37 tratam da `FileNotFoundException`, que ocorre se o arquivo não existir e um novo arquivo não puder ser criado. Essa exceção também pode acontecer se houver um erro ao *abrir* o arquivo. Em ambas as rotinas de tratamento de exceção, chamamos o método `static` de `System.exit` e passamos o valor `1`. Esse método encerra o aplicativo. Um argumento de `0` para o método `exit` indica terminação *bem-sucedida* do programa. Um valor diferente de zero, como `1` neste exemplo, normalmente indica que ocorreu um erro. Esse valor é passado à janela de comando que executou o programa. O argumento é útil se o programa for executado a partir de um **arquivo em lote** em sistemas Windows ou de um **script de shell** nos sistemas UNIX/Linux/Mac OS X. Os arquivos em lote e scripts de shell oferecem uma maneira conveniente para executar vários programas em sequência. Quando o primeiro programa encerra, o próximo inicia a execução. É possível utilizar o argumento para o método `exit` em um arquivo em lote ou script de shell a fim de determinar se outros programas devem ser executados. Para mais informações sobre arquivos em lote ou scripts de shell, veja a documentação do seu sistema operacional.

O método `addRecords` (linhas 41 a 69) pede que o usuário insira os vários campos para cada registro ou a sequência de teclas de fim de arquivo quando a entrada de dados estiver completa. A Figura 15.4 lista as combinações de teclas para inserir o fim de arquivo para vários sistemas de computador.

Sistema operacional	Combinação de teclas
UNIX/linux/Mac OS X	*<Enter> <Ctrl> d*
Windows	*<Ctrl> z*

Figura 15.4 | Combinações de chaves de fim de arquivo.

As linhas 44 a 46 solicitam uma entrada ao usuário. A linha 48 utiliza o método `Scanner` `hasNext` para determinar se a combinação de teclas de fim de arquivo foi inserida. O loop executa até que `hasNext` encontre o fim de arquivo.

As linhas 53 e 54 usam um `Scanner` para ler os dados do usuário, então geram uma saída deles como um registro usando o `Formatter`. Cada método de entrada `Scanner` lança uma **`NoSuchElementException`** (tratada nas linhas 61 a 65) se os dados estiverem no formato errado (por exemplo, uma `String` quando um `int` é esperado) ou se não houver mais dados para serem inseridos. As informações do registro são enviadas para a saída com o método **`format`**, que pode realizar uma formatação idêntica à do método `System.out.printf` usado extensivamente nos capítulos anteriores. O método `format` envia uma `String` formatada para o destino da saída do objeto `Formatter` — o arquivo `clients.txt`. A string de formato `"%d %s %s %.2f%n"` indica que o registro atual será armazenado como um inteiro (o número de conta) seguido por uma `String` (o nome), outra `String` (o sobrenome) e um valor de ponto flutuante (o saldo). Cada informação é separada da seguinte por um espaço, e a saída do valor de double (o saldo) é gerada com dois dígitos à direita do ponto de fração decimal (como indicado pelo `.2` em `%.2f`). Os dados no arquivo de texto podem ser visualizados com um editor ou recuperados mais tarde por um programa projetado para ler o arquivo (Seção 15.4.2).

Quando as linhas 66 a 68 são executadas, se o objeto `Formatter` estiver fechado, uma **`FormatterClosedException`** será lançada. Essa exceção é tratada nas linhas 76 a 80. [*Observação*: você também pode gerar saída de dados para um arquivo de texto utilizando a classe **`java.io.PrintWriter`**, que fornece os métodos `format` e `printf` para gerar saída de dados formatados.]

As linhas 93 a 97 declaram o método `closeFile`, que fecha o `Formatter` e o arquivo de saída subjacente. A linha 96 fecha o objeto simplesmente chamando o método **`close`**. Se método `close` não for chamado explicitamente, o sistema operacional em geral fechará o arquivo quando a execução do programa terminar — esse é um exemplo de "faxina" do sistema operacional. Mas você sempre deve fechar explicitamente um arquivo quando ele não mais é necessário.

Saída de exemplo

Os dados de exemplo para esse aplicativo são mostrados na Figura 15.5. Na saída de exemplo, o usuário insere informações para cinco contas, então insere o fim de arquivo a fim de sinalizar que a entrada de dados está concluída. A saída de exemplo não mostra como os registros de dados na verdade aparecem no arquivo. Na próxima seção, para verificar se o arquivo foi criado com sucesso, apresentaremos um programa que lê o arquivo e imprime seu conteúdo. Como esse é um arquivo de texto, você também pode verificar as informações simplesmente abrindo-o em um editor de textos.

Dados de exemplo			
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

Figura 15.5 | Dados de exemplo para o programa na Figura 15.3.

15.4.2 Lendo dados a partir de um arquivo de texto de acesso sequencial

Os dados são armazenados em arquivos de modo que possam ser recuperados para processamento quando necessário. A Seção 15.4.1 demonstrou como criar um arquivo de acesso sequencial. Esta seção mostrará como ler dados sequencialmente em um arquivo de texto. Demonstramos como a classe `Scanner` pode ser utilizada para inserir dados a partir de um arquivo, em vez de utilizar o teclado. O aplicativo (Figura 15.6) lê os registros a partir do arquivo `"clients.txt"` criado pelo aplicativo da Seção 15.4.1 e exibe o conteúdo. A linha 13 declara um `Scanner` que será usado para recuperar a entrada do arquivo.

```
1  // Figura 15.6: ReadTextFile.java
2  // Esse programa lê um arquivo de texto e exibe 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 ReadTextFile
12 {
```

continua

continuação

```
13       private static Scanner input;
14
15       public static void main(String[] args)
16       {
17          openFile();
18          readRecords();
19          closeFile();
20       }
21
22       // abre o arquivo clients.txt
23       public static void openFile()
24       {
25          try
26          {
27             input = new Scanner(Paths.get("clients.txt"));
28          }
29          catch (IOException ioException)
30          {
31             System.err.println("Error opening file. Terminating.");
32             System.exit(1);
33          }
34       }
35
36       // lê o registro no arquivo
37       public static void readRecords()
38       {
39          System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
40             "First Name", "Last Name", "Balance");
41
42          try
43          {
44             while (input.hasNext()) // enquanto houver mais para ler
45             {
46                // exibe o conteúdo de registro
47                System.out.printf("%-10d%-12s%-12s%10.2f%n", input.nextInt(),
48                   input.next(), input.next(), input.nextDouble());
49             }
50          }
51          catch (NoSuchElementException elementException)
52          {
53             System.err.println("File improperly formed. Terminating.");
54          }
55          catch (IllegalStateException stateException)
56          {
57             System.err.println("Error reading from file. Terminating.");
58          }
59       } // fim do método readRecords
60
61       // fecha o arquivo e termina o aplicativo
62       public static void closeFile()
63       {
64          if (input != null)
65             input.close();
66       }
67    } // fim da classe ReadTextFile
```

```
Account   First Name  Last Name     Balance
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
```

Figura 15.6 | Leitura de arquivo sequencial utilizando um `Scanner`.

O método `openFile` (linhas 23 a 34) abre o arquivo para leitura instanciando um objeto `Scanner` na linha 27. Passamos um objeto `Path` para o construtor, que especifica que o objeto `Scanner` irá ler a partir do arquivo `"clients.txt"` localizado no diretório em que o aplicativo é executado. Se o arquivo não puder ser localizado, ocorrerá uma `IOException`. O tratamento de exceção ocorre nas linhas 29 a 33.

O método `readRecords` (linhas 37 a 59) lê e exibe os registros do arquivo. As linhas 39 e 40 exibem os cabeçalhos das colunas na saída do aplicativo. As linhas 44 a 49 leem e mostram os dados do arquivo até que o *marcador de fim de arquivo* é alcançado (nesse caso, o método `hasNext` retornará `false` na linha 44). As linhas 47 e 48 utilizam os métodos `Scanner` `nextInt`, `next` e `nextDouble` para inserir um `int` (o número de conta), duas `Strings` (nomes e sobrenomes) e um valor de `double` (o saldo). Cada registro é uma linha de dados no arquivo. Se as informações no arquivo não estão adequadamente formatadas (por exemplo, há um sobrenome onde deveria haver um saldo), ocorre uma `NoSuchElementException` quando o registro é inserido. Essa exceção é tratada nas linhas 51 a 54. Se o `Scanner` for fechado antes de os dados serem inseridos, há uma **`IllegalStateException`** (tratada nas linhas 55 a 58). Observe na string de formato na linha 47 que o número de conta, nome e sobrenome são alinhados à esquerda, enquanto o saldo é alinhado à direita e enviado para a saída com dois dígitos de precisão. Cada iteração do loop insere uma linha de texto no arquivo de texto, que representa um registro. As linhas 62 a 66 definem o método `closeFile`, que fecha a `Scanner`.

15.4.3 Estudo de caso: um programa de consulta de crédito

Para recuperar sequencialmente dados de um arquivo, os programas começam no início do arquivo e leem *todos* os dados de maneira consecutiva até que as informações desejadas sejam encontradas. Talvez seja necessário processar sequencialmente o arquivo várias vezes (a partir do início dele) durante a execução de um programa. A classe `Scanner` *não* permite reposicionamento para o início do arquivo. Se for preciso ler o arquivo de novo, o programa deverá *fechá-lo* e *reabri-lo*.

O programa nas figuras 15.7 e 15.8 permite que um gerente de crédito obtenha listas de clientes com *saldo zero* (isto é, clientes que não devem nada à empresa), clientes com *saldos credores* (isto é, clientes aos quais a empresa deve dinheiro) e clientes com *saldos devedores* (isto é, clientes que devem à empresa por bens e serviços recebidos). Um saldo credor é um valor monetário *negativo,* e um saldo devedor, um valor *positivo*.

MenuOption enum

Começaremos criando um tipo `enum` (Figura 15.7) para definir as diferentes opções de menu que o gerente de crédito terá — isso é necessário se você precisar fornecer valores específicos para as constantes `enum`. As opções e seus valores estão listados nas linhas 7 a 10.

```
1  // Figura 15.7: MenuOption.java
2  // tipo enum para as opções do programa de consulta de crédito.
3
4  public enum MenuOption
5  {
6     // declara o conteúdo do tipo enum
7     ZERO_BALANCE(1),
8     CREDIT_BALANCE(2),
9     DEBIT_BALANCE(3),
10    END(4);
11
12    private final int value; // opção atual de menu
13
14    // construtor
15    private MenuOption(int value)
16    {
17       this.value = value;
18    }
19 } // fim do enum de MenuOption
```

Figura 15.7 | Tipo `enum` para as opções do menu do programa de consulta de crédito.

Classe CreditInquiry

A Figura 15.8 contém a funcionalidade para o programa de consulta de crédito. O programa exibe um menu de texto e permite ao gerente de crédito inserir uma de três opções para obter informações de crédito:

- A opção 1 (`ZERO_BALANCE`) exibe as contas com saldo zero.
- A opção 2 (`CREDIT_BALANCE`) exibe as contas com saldos credores.
- A opção 3 (`DEBIT_BALANCE`) exibe as contas com saldos devedores.
- A opção 4 (`END`) encerra a execução do programa.

```
1  // Figura 15.8: CreditInquiry.java
2  // Esse programa lê um arquivo sequencialmente e exibe o
3  // conteúdo baseado no tipo de conta que o usuário solicita
4  // (saldo credor, saldo devedor ou saldo zero).
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 CreditInquiry
12 {
13    private final static MenuOption[] choices = MenuOption.values();
14
15    public static void main(String[] args)
16    {
17       // obtém a solicitação do usuário (por exemplo, saldo zero, credor ou devedor)
18       MenuOption accountType = getRequest();
19
20       while (accountType != MenuOption.END)
21       {
22          switch (accountType)
23          {
24             case ZERO_BALANCE:
25                System.out.printf("%nAccounts with zero balances:%n");
26                break;
27             case CREDIT_BALANCE:
28                System.out.printf("%nAccounts with credit balances:%n");
29                break;
30             case DEBIT_BALANCE:
31                System.out.printf("%nAccounts with debit balances:%n");
32                break;
33          }
34
35          readRecords(accountType);
36          accountType = getRequest(); // obtém a solicitação do usuário
37       }
38    }
39
40    // obtém a solicitação do usuário
41    private static MenuOption getRequest()
42    {
43       int request = 4;
44
45       // exibe opções de solicitação
46       System.out.printf("%nEnter request%n%s%n%s%n%s%n%s%n",
47          " 1 - List accounts with zero balances",
48          " 2 - List accounts with credit balances",
49          " 3 - List accounts with debit balances",
50          " 4 - Terminate program");
51
52       try
53       {
54          Scanner input = new Scanner(System.in);
55
56          do // insere a solicitação de usuário
57          {
58             System.out.printf("%n? ");
59             request = input.nextInt();
60          } while ((request < 1) || (request > 4));
61       }
62       catch (NoSuchElementException noSuchElementException)
63       {
64          System.err.println("Invalid input. Terminating.");
65       }
66
67       return choices[request - 1]; // retorna o valor enum da opção
68    }
69
70    // lê registros de arquivo e exibe somente os registros do tipo apropriado
```

continua

continuação

```
71       private static void readRecords(MenuOption accountType)
72       {
73          // abre o arquivo e processa o conteúdo
74          try (Scanner input = new Scanner(Paths.get("clients.txt")))
75          {
76             while (input.hasNext()) // mais dados para ler
77             {
78                int accountNumber = input.nextInt();
79                String firstName = input.next();
80                String lastName = input.next();
81                double balance = input.nextDouble();
82
83                // se o tipo for a conta adequada, exibe o registro
84                if (shouldDisplay(accountType, balance))
85                   System.out.printf("%-10d%-12s%-12s%10.2f%n", accountNumber,
86                      firstName, lastName, balance);
87                else
88                   input.nextLine(); // descarta o restante do registro atual
89             }
90          }
91          catch (NoSuchElementException |
92             IllegalStateException | IOException e)
93          {
94             System.err.println("Error processing file. Terminating.");
95             System.exit(1);
96          }
97       } // fim do método readRecords
98
99       // utiliza o tipo de registro para determinar se registro deve ser exibido
100      private static boolean shouldDisplay(
101         MenuOption accountType, double balance)
102      {
103         if ((accountType == MenuOption.CREDIT_BALANCE) && (balance < 0))
104            return true;
105         else if ((accountType == MenuOption.DEBIT_BALANCE) && (balance > 0))
106            return true;
107         else if ((accountType == MenuOption.ZERO_BALANCE) && (balance == 0))
108            return true;
109
110         return false;
111      }
112   } // fim da classe CreditInquiry
```

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - Terminate program

? 1

Accounts with zero balances:
300      Pam        White           0.00

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - Terminate program

? 2

Accounts with credit balances:
200      Steve      Green        -345.67
400      Sam        Red          -42.16
```

continua

continuação

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - Terminate program

? 3

Accounts with debit balances:
100      Bob       Blue          24.98
500      Sue       Yellow       224.62

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - Terminate program

? 4
```

Figura 15.8 | Programa de consulta de crédito.

As informações do registro são coletadas lendo o arquivo e determinando se cada registro atende aos critérios para o tipo de conta selecionado. A linha 18 em `main` chama o método `getRequest` (linhas 41 a 68) para exibir as opções de menu, converte o número digitado pelo usuário em um `MenuOption` e armazena o resultado na variável `MenuOption accountType`. As linhas 20 a 37 fazem um loop até que o usuário especifique que o programa deve encerrar. As linhas 22 a 33 mostram um cabeçalho para o conjunto atual de registros a ser gerado na tela. A linha 35 chama o método `readRecords` (linhas 71 a 97), que faz o loop pelo arquivo e lê cada registro.

O método `readRecords` usa uma instrução `try` com recursos (introduzida na Seção 11.12) para criar um `Scanner` que abre o arquivo para leitura (linha 74) — lembre-se de que `try` com recursos fechará o(s) recurso(s) quando o bloco `try` encerra com sucesso ou por causa de uma exceção. O arquivo será aberto para leitura com um novo objeto `Scanner` sempre que `readRecords` é chamado, assim podemos ler novamente a partir do início do arquivo. As linhas 78 a 81 leem um registro. A linha 84 chama o método `shouldDisplay` (linhas 100 a 111) para determinar se o registro atual satisfaz o tipo de conta solicitado. Se `shouldDisplay` retornar `true`, o programa exibirá as informações de conta. Quando o *marcador de fim de arquivo* é alcançado, o loop termina e a instrução `try` com recursos fecha o `Scanner` e o arquivo. Depois que todos os registros foram lidos, o controle retorna ao `main` e `getRequest` é mais uma vez chamado (linha 36) para recuperar a próxima opção de menu do usuário.

15.4.4 Atualizando arquivos de acesso sequencial

Os dados em muitos arquivos sequenciais não podem ser modificados sem o risco de destruir outros. Por exemplo, se for necessário alterar o nome "`White`" para "`Worthington`", o nome antigo simplesmente não poderá ser sobrescrito, porque o novo nome requer mais espaço. O registro para `White` foi gravado no arquivo como

```
300 Pam White 0.00
```

Se o registro fosse regravado começando no mesmo local no arquivo utilizando o novo nome, ele seria

```
300 Pam Worthington 0.00
```

O novo registro é maior (tem mais caracteres) que o original. "`Worthington`" sobrescreveria o "`0.00`" no registro atual, e os caracteres além do segundo "`o`" nesse nome sobrescreveriam o início do próximo registro sequencial no arquivo. O problema aqui é que os campos em um arquivo de texto — e consequentemente os registros — podem variar de tamanho. Por exemplo, 7, 14, –117, 2074 e 27383 são todos `int`s armazenados no mesmo número de bytes (4) internamente, mas são campos com diferentes tamanhos quando escritos em um arquivo como texto. Portanto, os registros em um arquivo de acesso sequencial em geral não são atualizados no local; em vez disso, todo o arquivo é regravado. Para fazer a alteração no nome anterior, os registros antes de `300 Pam White 0.00` seriam copiados para um novo arquivo, o novo registro (que pode ter um tamanho diferente daquele que ele substitui) seria gravado e os registros depois de `300 Pam White 0.00` seriam copiados para o novo arquivo. Regravar todo o arquivo não é econômico para atualizar um único registro, mas razoável se um número substancial de registros precisar ser atualizado.

15.5 Serialização de objeto

Na Seção 15.4, demonstramos como gravar os campos individuais de um registro em um arquivo como texto, e como ler esses campos. Quando a saída dos dados foi enviada para o disco, algumas informações foram perdidas, como o tipo de cada valor. Por exemplo, se o valor `"3"` for lido a partir de um arquivo, não há como dizer se ele veio de um `int`, uma `String` ou um `double`. Temos apenas dados, não informações de tipo, em um disco.

Às vezes queremos ler ou gravar um objeto em um arquivo ou em uma conexão de rede. O Java fornece a **serialização de objetos** para esse propósito. Um **objeto serializado** é representado como uma sequência de bytes que inclui os dados do objeto, bem como as informações sobre o tipo dele e a natureza dos dados armazenados nele. Depois que um objeto serializado foi gravado em um arquivo, ele pode ser lido a partir do arquivo e **desserializado** — isto é, as informações dos tipos e bytes que representam o objeto e seus dados podem ser utilizadas para recriar o objeto na memória.

Classes `ObjectInputStream` e `ObjectOutputStream`

As classes `ObjectInputStream` e `ObjectOutputStream` (pacote `java.io`), que, respectivamente, implementam as interfaces `ObjectInput` e `ObjectOutput`, permitem que objetos inteiros sejam lidos ou gravados em um fluxo (possivelmente um arquivo). Para usar a serialização com arquivos, inicializamos os objetos `ObjectInputStream` e `ObjectOutputStream` com objetos de fluxo que leem e gravam em arquivos. Esse tipo de inicialização de objetos de fluxo com outros objetos de fluxo é chamado, às vezes, de **empacotamento** — o novo objeto de fluxo em processo de criação empacota o objeto de fluxo especificado como um argumento do construtor.

As classes `ObjectInputStream` e `ObjectOutputStream` simplesmente leem e gravam a representação baseada em bytes dos objetos — elas não sabem onde ler os bytes ou gravá-los. O objeto de fluxo passado para o construtor `ObjectInputStream` fornece os bytes que o `ObjectInputStream` converte em objetos. Da mesma forma, o objeto de fluxo passado para o construtor `ObjectOutputStream` recebe a representação baseada em bytes do objeto que o `ObjectOutputStream` produz e grava os bytes no destino especificado (por exemplo, um arquivo, uma conexão de rede etc.).

Interfaces `ObjectOutput` e `ObjectInput`

A interface `ObjectOutput` contém o método **`writeObject`**, que recebe um `Object` como um argumento e grava suas informações em um `OutputStream`. Uma classe que implementa a interface `ObjectOutput` (como `ObjectOutputStream`) declara esse método e garante que o objeto que é gerado trabalha a interface `Serializable` (discutida em breve). Da mesma forma, a interface `ObjectInput` contém o método **`readObject`**, que lê e retorna uma referência a um `Object` a partir de um `InputStream`. Depois que um objeto foi lido, podemos fazer uma coerção da sua referência para o tipo real do objeto. Como você verá no Capítulo 28 (em inglês, na Sala Virtual), aplicativos que se comunicam por uma rede, como a internet, também podem transmitir objetos por ela.

15.5.1 Criando um arquivo de acesso sequencial com a serialização de objeto

Esta seção e a Seção 15.5.2 criam e manipulam arquivos de acesso sequencial usando a serialização de objeto. A serialização de objeto que mostramos aqui é realizada com fluxos baseados em bytes, assim arquivos sequenciais criados e manipulados serão *arquivos binários*. Lembre-se de que, em geral, arquivos binários não podem ser visualizados nos editores de texto padrão. Por essa razão, escrevemos um aplicativo separado que sabe ler e exibir objetos serializados. Iniciamos criando e gravando objetos serializados em um arquivo de acesso sequencial. O exemplo é semelhante àquele na Seção 15.4, portanto, focalizaremos apenas os novos recursos.

Definindo a classe `Account`

Começamos definindo a classe `Account` (Figura 15.9), que encapsula as informações sobre o registro do cliente usadas pelos exemplos de serialização. Esses exemplos e a classe `Account` estão localizados no diretório `SerializationApps` com os exemplos do capítulo. Isso permite que a classe `Account` seja utilizada pelos dois exemplos, porque seus arquivos são definidos no mesmo pacote padrão. A classe `Account` contém as variáveis de instância `private` `account`, `firstName`, `lastName` e `balance` (linhas 7 a 10), além dos métodos *set* e *get* para acessar essas variáveis de instância. Embora os métodos *set* não validem os dados nesse exemplo, eles devem fazer isso em um sistema de produção robusto. A classe `Account` implementa a interface **`Serializable`** (linha 5), o que permite que objetos dessa classe sejam *serializados* e *desserializados* com `ObjectOutputStream`s e `ObjectInputStream`s, respectivamente. A interface `Serializable` é uma **interface de tags**. Essa interface *não* contém nenhum método. Uma classe que implementa `Serializable` é marcada com *tags* como um objeto `Serializable`. Isso é importante, pois um `ObjectOutputStream` *não* enviará para a saída um objeto, a menos que ele *seja um* objeto `Serializable`, que é o caso para qualquer um de uma classe que implementa `Serializable`.

```
1  // Figura 15.9: Account.java
2  // Classe Account serializável para armazenar registros como objetos.
3  import java.io.Serializable;
4
5  public class Account implements Serializable
6  {
7     private int account;
8     private String firstName;
9     private String lastName;
10    private double balance;
11
12    // inicializa uma Account com valores padrão
```

continua

continuação

```
13       public Account()
14       {
15          this(0, "", "", 0.0); // chama outro construtor
16       }
17
18       // inicializa uma Account com os valores fornecidos
19       public Account(int account, String firstName,
20          String lastName, double balance)
21       {
22          this.account = account;
23          this.firstName = firstName;
24          this.lastName = lastName;
25          this.balance = balance;
26       }
27
28       // configura o número de conta
29       public void setAccount(int acct)
30       {
31          this.account = account;
32       }
33
34       // obtém número de conta
35       public int getAccount()
36       {
37          return account;
38       }
39
40       // configura o nome
41       public void setFirstName(String firstName)
42       {
43          this.firstName = firstName;
44       }
45
46       // obtém o nome
47       public String getFirstName()
48       {
49          return firstName;
50       }
51
52       // configura o sobrenome
53       public void setLastName(String lastName)
54       {
55          this.lastName = lastName;
56       }
57
58       // obtém o sobrenome
59       public String getLastName()
60       {
61          return lastName;
62       }
63
64       // configura saldo
65       public void setBalance(double balance)
66       {
67          this.balance = balance;
68       }
69
70       // obtém saldo
71       public double getBalance()
72       {
73          return balance;
74       }
75    } // fim da classe Account
```

Figura 15.9 | Classe `Account` para objetos serializáveis.

Em uma classe `Serializable`, cada variável de instância deve ser `Serializable`. Variáveis de instância não `Serializable` devem ser declaradas **`transient`** para indicar que elas devem ser ignoradas durante o processo de serialização. *Por padrão, todas as variáveis de tipo primitivo são serializáveis.* Para variáveis de tipo por referência, você precisa verificar a documentação da classe (e possivelmente suas superclasses) a fim de garantir que o tipo é `Serializable`. Por exemplo, `String`s são `Serializable`. Por padrão, os arrays *são* serializáveis; mas em um array de tipos por referência, os objetos referenciados talvez *não* o sejam. A classe `Account` contém os membros de dados `private` `account`, `firstName`, `lastName` e `balance` — todos eles são `Serializable`. Essa classe também fornece os métodos `public` *get* e *set* para acessar os campos `private`.

Gravando objetos serializados em um arquivo de acesso sequencial

Agora vamos discutir o código que cria o arquivo de acesso sequencial (Figura 15.10). Aqui, nos concentraremos apenas nos novos conceitos. Para abrir o arquivo, a linha 27 chama o método `Files` `static` **`newOutputStream`**, que recebe um `Path` especificando o arquivo a abrir e, se este existir, retorna um `OutputStream` que pode ser usado para gravar no arquivo. Arquivos existentes que são abertos para a saída dessa maneira são *truncados*. Não há nenhuma extensão de nome de arquivo padrão para aqueles que armazenam objetos serializados, portanto, escolhemos `.ser`.

```
1  // Figura 15.10: CreateSequentialFile.java
2  // Gravando objetos sequencialmente em um arquivo com a classe 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 CreateSequentialFile
11 {
12    private static ObjectOutputStream output; // gera saída dos dados no arquivo
13
14    public static void main(String[] args)
15    {
16       openFile();
17       addRecords();
18       closeFile();
19    }
20
21    // abre o arquivo clients.ser
22    public static void openFile()
23    {
24       try
25       {
26          output = new ObjectOutputStream(
27             Files.newOutputStream(Paths.get("clients.ser")));
28       }
29       catch (IOException ioException)
30       {
31          System.err.println("Error opening file. Terminating.");
32          System.exit(1); // termina o programa
33       }
34    }
35
36    // adiciona registros ao arquivo
37    public static void addRecords()
38    {
39       Scanner input = new Scanner(System.in);
40
41       System.out.printf("%s%n%s%n? ",
42          "Enter account number, first name, last name and balance.",
43          "Enter end-of-file indicator to end input.");
44
45       while (input.hasNext()) // faz um loop até o indicador de fim de arquivo
46       {
47          try
48          {
49             // cria novo registro; esse exemplo supõe uma entrada válida
```

continua

continuação

```
50                Account record = new Account(input.nextInt(),
51                   input.next(), input.next(), input.nextDouble());
52
53                // serializa o objeto de registro em um arquivo
54                output.writeObject(record);
55             }
56             catch (NoSuchElementException elementException)
57             {
58                System.err.println("Invalid input. Please try again.");
59                input.nextLine(); // descarta entrada para o usuário tentar de novo
60             }
61             catch (IOException ioException)
62             {
63                System.err.println("Error writing to file. Terminating.");
64                break;
65             }
66
67             System.out.print("? ");
68          }
69       }
70
71       // fecha o arquivo e termina o aplicativo
72       public static void closeFile()
73       {
74          try
75          {
76             if (output != null)
77                output.close();
78          }
79          catch (IOException ioException)
80          {
81             System.err.println("Error closing file. Terminating.");
82          }
83       }
84    } // fim da classe CreateSequentialFile
```

```
Enter account number, first name, last name and balance.
Enter end-of-file indicator to end input.
? 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
```

Figura 15.10 | Arquivo sequencial criado com `ObjectOutputStream`.

A classe `OutputStream` fornece os métodos a fim de enviar para a saída os arrays `byte` e os `bytes` individuais, mas queremos gravar *objetos* em um arquivo. Por essa razão, as linhas 26 e 27 passam o `InputStream` para o construtor da classe `ObjectInputStream`, que *empacota* o `OutputStream` em um `ObjectOutputStream`. O objeto `ObjectOutputStream` utiliza o `OutputStream` para gravar no arquivo os bytes que representam objetos inteiros. As linhas 26 e 27 talvez lancem uma **`IOException`** se um problema ocorrer ao abrir o arquivo (por exemplo, quando um arquivo é aberto para gravação em uma unidade com espaço insuficiente ou quando um arquivo de leitura é aberto para gravação). Se isso ocorrer, o programa exibirá uma mensagem de erro (linhas 29 a 33). Se nenhuma exceção ocorrer, o arquivo é aberto e a variável `output` pode ser utilizada para gravar objetos nele.

Esse programa supõe que os dados foram inseridos corretamente e na ordem de número de registro adequada. O método `addRecords` (linhas 37 a 69) realiza a operação de gravação. As linhas 50 e 51 criam um objeto `Account` a partir dos dados inseridos pelo usuário. A linha 54 chama o método `ObjectOutputStream` `writeObject` para gravar o objeto `record` no arquivo de saída. Apenas uma instrução é necessária para gravar o objeto *inteiro*.

O método `closeFile` (linhas 72 a 83) chama o método `ObjectOutputStream` **`close`** em `output` para fechar `ObjectOutputStream` *e seu* `OutputStream` subjacente. A chamada para o método `close` está contida em um bloco `try` porque `close` lança uma `IOException` se o arquivo não puder ser fechado adequadamente. Ao utilizar fluxos *empacotados*, fechar o fluxo externo *também* fecha o fluxo empacotado.

Na elaboração de exemplo para o programa na Figura 15.10, inserimos informações para cinco contas — as mesmas informações mostradas na Figura 15.5. O programa não mostra como os registros de dados na verdade aparecem no arquivo. Lembre-se de que agora estamos utilizando *arquivos binários*, que não são humanamente legíveis. Para verificar se o arquivo foi criado com sucesso, a próxima seção apresenta um programa para ler o conteúdo do arquivo.

15.5.2 Lendo e desserializando dados a partir de um arquivo de acesso sequencial

A seção anterior mostrou como criar um arquivo de acesso sequencial utilizando a serialização de objetos. Nesta seção, discutiremos como *ler dados serializados* sequencialmente a partir de um arquivo.

O programa na Figura 15.11 lê registros de um arquivo criado pelo programa na Seção 15.5.1 e exibe o conteúdo. Esse programa abre o arquivo para entrada chamando o método `Files static` **`newInputStream`**, que recebe um `Path` especificando o arquivo a abrir e, se ele existir, retorna um `InputStream` que pode ser usado para ler a partir do arquivo. Na Figura 15.10, os objetos no arquivo foram gravados com um objeto `ObjectOutputStream`. Os dados devem ser lidos do arquivo no mesmo formato em que foram gravados. Portanto, usamos um `ObjectInputStream` para *empacotar* um `InputStream` (linhas 26 e 27). Se nenhuma exceção ocorrer ao abrir o arquivo, a variável `input` pode ser usada para ler objetos dele.

```
1  // Figura 15.11: ReadSequentialFile.java
2  // Lendo um arquivo dos objetos sequencialmente com ObjectInputStream
3  // e exibindo cada registro.
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 ReadSequentialFile
11 {
12    private static ObjectInputStream input;
13
14    public static void main(String[] args)
15    {
16       openFile();
17       readRecords();
18       closeFile();
19    }
20
21    // permite que o usuário selecione o arquivo a abrir
22    public static void openFile()
23    {
24       try // abre o arquivo
25       {
26          input = new ObjectInputStream(
27             Files.newInputStream(Paths.get("clients.ser")));
28       }
29       catch (IOException ioException)
30       {
31          System.err.println("Error opening file.");
32          System.exit(1);
33       }
34    }
35
36    // lê o registro no arquivo
37    public static void readRecords()
38    {
39       System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
40          "First Name", "Last Name", "Balance");
41
42       try
43       {
44          while (true) // faz um loop até ocorrer uma EOFException
45          {
46             Account record = (Account) input.readObject();
47
48             // exibe o conteúdo de registro
```

continua

continuação

```
49                System.out.printf("%-10d%-12s%-12s%10.2f%n",
50                   record.getAccount(), record.getFirstName(),
51                   record.getLastName(), record.getBalance());
52             }
53          }
54          catch (EOFException endOfFileException)
55          {
56             System.out.printf("%No more records%n");
57          }
58          catch (ClassNotFoundException classNotFoundException)
59          {
60             System.err.println("Invalid object type. Terminating.");
61          }
62          catch (IOException ioException)
63          {
64             System.err.println("Error reading from file. Terminating.");
65          }
66       } // fim do método readRecords
67
68       // fecha o arquivo e termina o aplicativo
69       public static void closeFile()
70       {
71          try
72          {
73             if (input != null)
74                input.close();
75          }
76          catch (IOException ioException)
77          {
78             System.err.println("Error closing file. Terminating.");
79             System.exit(1);
80          }
81       }
82    } // fim da classe ReadSequentialFile
```

```
Account  First Name  Last Name     Balance
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 more records
```

Figura 15.11 | Lendo um arquivo de objetos sequencialmente com `ObjectInputStream` e exibindo cada registro.

O programa lê registros do arquivo no método `readRecords` (linhas 37 a 66). A linha 46 chama o método `ObjectInputStream` `readObject` para ler um `Object` do arquivo. Para utilizar os métodos específicos `Account`, fazemos um *downcast* do `Object` retornado para o tipo `Account`. O método `readObject` lança uma **`EOFException`** (processada nas linhas 54 a 57) se ocorrer uma tentativa de leitura além do final do arquivo. O método `readObject` lança uma `ClassNotFoundException` se a classe do objeto que está sendo lido não puder ser localizada. Isso pode acontecer se o arquivo acessado em um computador não tiver essa classe.

Observação de engenharia de software 15.1

Esta seção apresentou a serialização de objetos e demonstrou as técnicas básicas desse processo. A serialização é um tema profundo com muitas complexidades e armadilhas. Antes de implementá-la em aplicativos de produção robustos, leia cuidadosamente a documentação Java on-line sobre a serialização de objetos.

15.6 Abrindo arquivos com `JFileChooser`

A classe **`JFileChooser`** exibe uma caixa de diálogo que permite ao usuário selecionar facilmente arquivos ou diretórios. Para demonstrar `JFileChooser`, aprimoramos o exemplo da Seção 15.3, como mostrado nas figuras 15.12 e 15.13. O exemplo agora contém uma interface gráfica com o usuário, mas continua a exibir os mesmos dados de antes. O construtor chama o método

`analyzePath` na linha 24. Esse método então chama o método `getFileOrDirectoryPath` na linha 31 para recuperar um objeto `Path` que representa o arquivo ou diretório selecionado.

O método `getFileOrDirectoryPath` (linhas 71 a 85 da Figura 15.12) cria um `JFileChooser` (linha 74). As linhas 75 e 76 chamam o método **`setFileSelectionMode`** para especificar o que o usuário pode selecionar no `fileChooser`. Para esse programa, utilizamos a constante `JFileChooser` `static` **`FILES_AND_DIRECTORIES`** a fim de indicar que arquivos e diretórios podem ser selecionados. Outras constantes `static` incluem **`FILES_ONLY`** (o padrão) e **`DIRECTORIES_ONLY`**.

A linha 77 chama o método **`showOpenDialog`** para exibir o diálogo de `JFileChooser` intitulado **Open**. O argumento `this` especifica a janela pai do diálogo de `JFileChooser`, que estabelece a posição do diálogo na tela. Se `null` for passado, o diálogo será exibido no centro da tela — caso contrário, ele é centralizado na janela do aplicativo (determinado pelo argumento `this`). Uma caixa de diálogo `JFileChooser` é uma *caixa de diálogo modal* que não permite que o usuário interaja com qualquer outra janela no programa até que o diálogo seja fechado. O usuário seleciona a unidade, o diretório ou o nome de arquivo e então clica em **Open**. O método `showOpenDialog` retorna um inteiro especificando em qual botão (**Open** ou **Cancel**) o usuário clicou para fechar o diálogo. A linha 48 testa se o usuário clicou em **Cancel** comparando o resultado com a constante `static` **`CANCEL_OPTION`**. Se forem iguais, o programa termina. A linha 84 chama o método `JFileChooser` **`getSelectedFile`** para recuperar um objeto de `File` (pacote `java.io`) que representa o arquivo ou diretório que o usuário selecionou, e então chama o método `File` **`toPath`** para retornar um objeto `Path`. O programa exibe, portanto, as informações sobre o arquivo ou diretório selecionado.

```
1  // Figura 15.12: JFileChooserDemo.java
2  // Demonstrando 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 JFileChooserDemo extends JFrame
15 {
16    private final JTextArea outputArea; // exibe o conteúdo do arquivo
17
18    // configura a GUI
19    public JFileChooserDemo() throws IOException
20    {
21       super("JFileChooser Demo");
22       outputArea = new JTextArea();
23       add(new JScrollPane(outputArea)); // outputArea é rolável
24       analyzePath(); // obtém o Path do usuário e exibe informações
25    }
26
27    // exibe informações sobre o arquivo ou diretório que o usuário especifica
28    public void analyzePath() throws IOException
29    {
30       // obtém o Path para o arquivo ou diretório selecionado pelo usuário
31       Path path = getFileOrDirectoryPath();
32
33       if (path != null && Files.exists(path)) // se existir, exibe as informações
34       {
35          // coleta as informações sobre o arquivo (ou diretório)
36          StringBuilder builder = new StringBuilder();
37          builder.append(String.format("%s:%n", path.getFileName()));
38          builder.append(String.format("%s a directory%n",
39             Files.isDirectory(path) ? "Is" : "Is not"));
40          builder.append(String.format("%s an absolute path%n",
41             path.isAbsolute() ? "Is" : "Is not"));
42          builder.append(String.format("Last modified: %s%n",
43             Files.getLastModifiedTime(path)));
44          builder.append(String.format("Size: %s%n", Files.size(path)));
```

continua

continuação

```
            builder.append(String.format("Path: %s%n", path));
            builder.append(String.format("Absolute path: %s%n",
               path.toAbsolutePath()));

            if (Files.isDirectory(path)) // listagem de diretório de saída
            {
               builder.append(String.format("%nDirectory contents:%n"));

               // objeto para iteração pelo conteúdo de um diretório
               DirectoryStream<Path> directoryStream =
                  Files.newDirectoryStream(path);

               for (Path p : directoryStream)
                  builder.append(String.format("%s%n", p));
            }

            outputArea.setText(builder.toString()); // exibe o conteúdo de String
         }
         else // Path não existe
         {
            JOptionPane.showMessageDialog(this, path.getFileName() +
               " does not exist.", "ERROR", JOptionPane.ERROR_MESSAGE);
         }
      } // fim do método analyzePath

      // permite que o usuário especifique o nome de arquivo ou diretório
      private Path getFileOrDirectoryPath()
      {
         // configura o diálogo permitindo a seleção de um arquivo ou diretório
         JFileChooser fileChooser = new JFileChooser();
         fileChooser.setFileSelectionMode(
            JFileChooser.FILES_AND_DIRECTORIES);
         int result = fileChooser.showOpenDialog(this);

         // se o usuário clicou no botão Cancel no diálogo, retorna
         if (result == JFileChooser.CANCEL_OPTION)
            System.exit(1);

         // retorna o Path representando o arquivo selecionado
         return fileChooser.getSelectedFile().toPath();
      }
   } // fim da classe JFileChooserDemo
```

Figura 15.12 | Demonstrando `JFileChooser`.

```
// Figura 15.13: JFileChooserTest.java
// Testa a classe JFileChooserDemo.
import java.io.IOException;
import javax.swing.JFrame;

public class JFileChooserTest
{
   public static void main(String[] args) throws IOException
   {
      JFileChooserDemo application = new JFileChooserDemo();
      application.setSize(400, 400);
      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      application.setVisible(true);
   }
} // fim da classe JFileChooserTest
```

continua

continuação

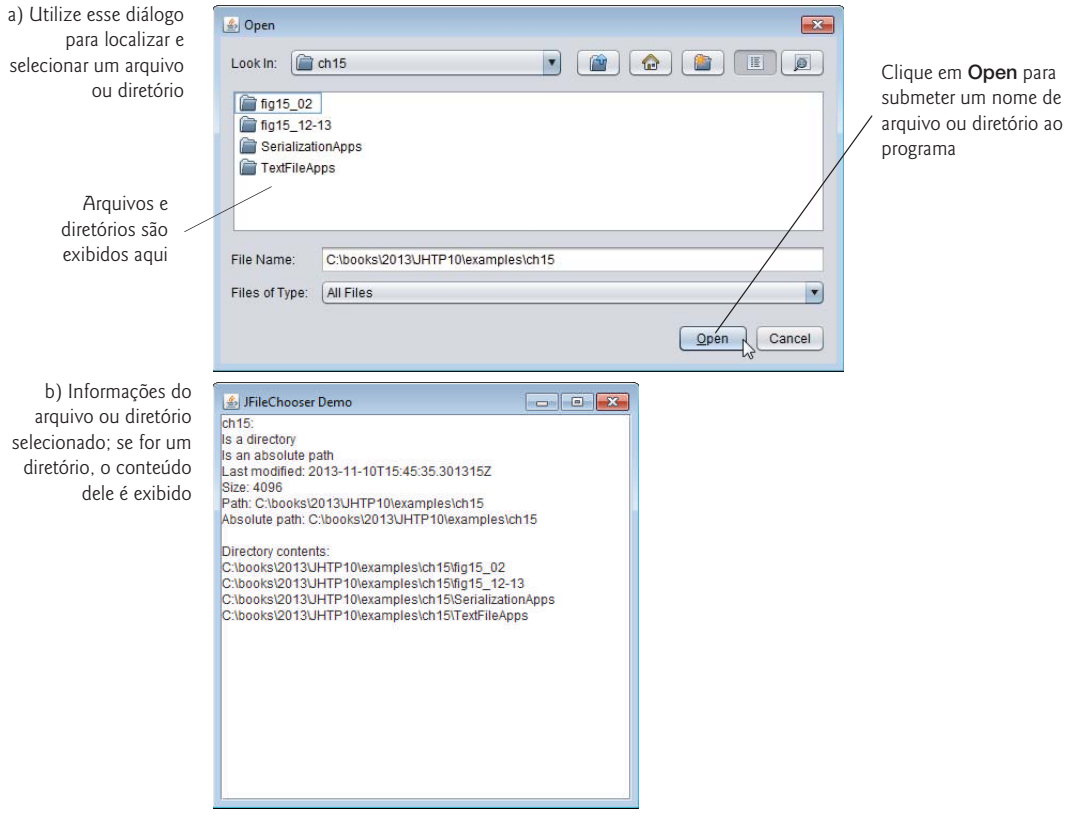


Figura 15.13 | Testando a classe `FileDemonstration`.

15.7 (Opcional) Classes `java.io` adicionais

Esta seção oferece uma visão geral das interfaces e classes (do pacote `java.io`) adicionais.

15.7.1 Interfaces e classes para entrada e saída baseadas em bytes

`InputStream` e `OutputStream` são classes `abstract` que declaram métodos para realizar entrada e saída baseadas em bytes, respectivamente.

Fluxos de pipe

Pipes são canais de comunicação sincronizados entre threads. Discutiremos threads no Capítulo 23. O Java fornece `PipedOutputStream` (uma subclasse de `OutputStream`) e `PipedInputStream` (uma subclasse de `InputStream`) para estabelecer pipes entre duas threads de um programa. Um thread envia dados a outro gravando em um `PipedOutputStream`. O thread alvo lê informações do pipe por um `PipedInputStream`.

Fluxos de filtro

Um `FilterInputStream` filtra um `InputStream`, e um `FilterOutputStream` filtra um `OutputStream`. A **filtragem** significa simplesmente que o fluxo do filtro fornece funcionalidades adicionais, como agregar bytes a unidades de tipos primitivos significativas. `FilterInputStream` e `FilterOutputStream` são em geral usados como superclasses, assim algumas das suas capacidades de filtragem são fornecidas pelas suas subclasses.

Um `PrintStream` (uma subclasse de `FilterOutputStream`) realiza a saída de texto para o fluxo especificado. Na verdade, até agora estamos usando a saída `PrintStream` por todo o livro — `System.out` e `System.err` são objetos `PrintStream`.

Fluxos de dados

Ler dados como bytes brutos é rápido, mas rudimentar. Normalmente, os programas leem os dados como agregados de bytes que formam `int`s, `float`s, `double`s e assim por diante. Programas Java podem utilizar várias classes para inserir e gerar saída de dados na forma agregada.

A interface `DataInput` descreve os métodos para ler tipos primitivos a partir de um fluxo de entrada. As classes **`DataInputStream`** e `RandomAccessFile` implementam essa interface para ler conjuntos de bytes e visualizá-los como valores de tipos primitivos. A interface `DataInput` inclui métodos como `readBoolean`, `readByte`, `readChar`, `readDouble`, `readFloat`, `readFully` (para arrays `byte`), `readInt`, `readLong`, `readShort`, `readUnsignedByte`, `readUnsignedShort`, `readUTF` (para ler caracteres Unicode codificados pelo Java — discutiremos a codificação UTF no Apêndice H, em inglês, na Sala Virtual do livro) e `skipBytes`.

A interface `DataOutput` descreve um conjunto de métodos para gravar tipos primitivos em um fluxo de saída. As classes **`DataOutputStream`** (uma subclasse de `FilterOutputStream`) e `RandomAccessFile` implementam, cada uma, essa interface para gravar valores de tipos primitivos como bytes. A interface `DataOutput` inclui versões sobrecarregadas do método `write` (para um `byte` ou um array de `byte`) e dos métodos `writeBoolean`, `writeByte`, `writeBytes`, `writeChar`, `writeChars` (para `String`s Unicode), `writeDouble`, `writeFloat`, `writeInt`, `writeLong`, `writeShort` e `writeUTF` (para gerar uma saída de texto modificado para Unicode).

Fluxos armazenados em buffer

Armazenamento em buffer (***buffering***) é uma técnica de aprimoramento do desempenho de E/S. Com um **`BufferedOutputStream`** (uma subclasse de `FilterOutputStream`), cada instrução de saída *não* necessariamente resulta em uma transferência física real de dados para o dispositivo de saída (uma operação lenta se comparada com as velocidades do processador e da memória principal). Em vez disso, cada operação de saída é dirigida para uma região na memória chamada **buffer**, que é grande o suficiente para armazenar os dados de muitas operações de saída. Então, a transferência real para o dispositivo de saída é realizada em uma grande **operação física de saída** toda vez que o buffer se enche. As operações de saída voltadas para o buffer de saída na memória são com frequência chamadas **operações lógicas de saída**. Com um `BufferedOutputStream`, um buffer parcialmente preenchido pode ser forçado a ir para o dispositivo a qualquer momento invocando o método **`flush`** do objeto de fluxo.

Utilizar o armazenamento em buffer pode aumentar de maneira significativa o desempenho de um aplicativo. Operações típicas de E/S são extremamente lentas se comparadas à velocidade de acesso aos dados na memória do computador. O armazenamento em buffer reduz o número de operações de E/S ao combinar primeiro saídas menores na memória. O número de operações físicas reais de E/S é pequeno se comparado ao número de solicitações de E/S emitidas pelo programa. Portanto, o programa que utiliza armazenamento em buffer é mais eficiente.

Dica de desempenho 15.1

E/S armazenada em buffer produz melhorias significativas de desempenho em relação a E/S não armazenada em buffer.

Com um **`BufferedInputStream`** (uma subclasse de `FilterInputStream`), muitos fragmentos ou trechos "lógicos" de dados de um arquivo são lidos como uma grande **operação física de entrada** em um buffer de memória. À medida que o programa solicita novos fragmentos de dados, eles são selecionados do buffer. (Esse procedimento é às vezes chamado de **operação lógica de entrada**.) Quando o buffer está vazio, a próxima operação física real de entrada do dispositivo de entrada é realizada para ler (*read*) o próximo grupo de trechos "lógicos" de dados. Portanto, o número de operações físicas reais de entrada é pequeno comparado com o número de solicitações de leitura emitido pelo programa.

Fluxos de array de `byte` baseados na memória

O fluxo de E/S do Java inclui capacidades para entrada e saída de arrays de `byte` na memória. Um `ByteArrayInputStream` (uma subclasse de `InputStream`) lê a partir de um array de `byte` na memória. Já um `ByteArrayOutputStream` (uma subclasse de `OutputStream`) gera saída para um array de `byte` na memória. Um dos usos de E/S de array de `byte` é a *validação de dados*. Um programa pode inserir uma linha inteira por vez do fluxo de entrada em um array de `byte`. Então, uma rotina de validação pode escrutinar o conteúdo do array de `byte` e corrigir os dados, se necessário. Por fim, o programa pode prosseguir para inserir a partir do array de `byte`, "sabendo" que os dados de entrada estão no formato adequado. Dar saída para um array de `byte` é uma boa maneira de tirar proveito das poderosas capacidades de formatação de fluxos de saída do Java. Por exemplo, os dados podem ser armazenados em um array de `byte`, utilizando a mesma formatação que será exibida em um momento posterior; podemos, assim, gerar a saída do array de `byte` em um arquivo para preservar a formatação.

Sequenciando entrada a partir de múltiplos fluxos

A `SequenceInputStream` (uma subclasse de `InputStream`) concatena logicamente vários `InputStream`s — o programa vê o grupo como um `InputStream` contínuo. Quando esse programa atinge o final de um fluxo de entrada, esse fluxo se fecha e o próximo fluxo na sequência se abre.

15.7.2 Interfaces e classes para entrada e saída baseadas em caracteres

Além dos fluxos baseados em bytes, o Java fornece as classes `Reader` e `Writer` `abstract`, que são fluxos baseados em caracteres como aqueles que você usou para processar arquivos de texto na Seção 15.4. A maioria dos fluxos baseados em bytes tem classes `Reader` ou `Writer` correspondentes concretas baseadas em caracteres.

Readers e Writers de buffer baseados em caracteres

As classes `BufferedReader` (uma subclasse de `Reader` `abstract`) e `BufferedWriter` (uma subclasse de `Writer` `abstract`) permitem armazenamento em buffer para fluxos baseados em caracteres. Lembre-se de que fluxos baseados em caracteres utilizam caracteres Unicode — esses fluxos podem processar dados em qualquer idioma que o conjunto de caracteres Unicode representa.

Readers e Writers de array char baseados na memória

As classes `CharArrayReader` e `CharArrayWriter` leem e gravam, respectivamente, um fluxo de caracteres em um array de `char`. Um `LineNumberReader` (uma subclasse de `BufferedReader`) é um fluxo de caracteres armazenado em buffer que monitora o número das linhas lidas — novas linhas, retornos de carro e combinações de retorno de carro incrementam a contagem de linhas. Monitorar os números da linha pode ser útil se o programa precisar informar ao leitor um erro em uma linha específica.

Readers e Writers de arquivos baseados em caracteres, pipe e string

Um `InputStream` pode ser convertido para um `Reader` por meio da classe `InputStreamReader`. Da mesma forma, um `OutputStream` pode ser convertido em um `Writer` por meio da classe `OutputStreamWriter`. A classe `FileReader` (uma subclasse de `InputStreamReader`) e a classe `FileWriter` (uma subclasse de `OutputStreamWriter`) leem e gravam caracteres em um arquivo, respectivamente. A classe `PipedReader` e a `PipedWriter` implementam fluxos de caracteres redirecionados (*piped*) para transferência de dados entre threads. A classe `StringReader` e `StringWriter` leem e gravam caracteres em `String`s, respectivamente. Um `PrintWriter` grava caracteres em um fluxo.

15.8 Conclusão

Neste capítulo, você aprendeu a manipular dados persistentes. Comparamos fluxos baseados em caracteres e fluxos baseados em bytes, além de introduzirmos várias classes dos pacotes `java.io` e `java.nio.file`. Você usou as classes `Files` e `Paths` e as interfaces `Path` e `DirectoryStream` para recuperar informações sobre arquivos e diretórios. Também empregou o processamento de arquivos de acesso sequencial a fim de manipular registros que são armazenados na ordem pelo campo de chave de registro. Você aprendeu as diferenças entre o processamento de arquivos de texto e a serialização de objetos, e utilizou a serialização para armazenar e recuperar objetos inteiros. O capítulo conclui com um pequeno exemplo de como usar uma caixa de diálogo `JFileChooser` para permitir que os usuários selecionem facilmente arquivos de uma GUI. O próximo capítulo discutirá as classes Java para manipular conjuntos de dados, como a classe `ArrayList`, que apresentamos na Seção 7.16.

Resumo

Seção 15.1 Introdução

- Computadores utilizam arquivos para armazenamento em longo prazo de grandes volumes de dados persistentes, mesmo depois de os programas que criaram os dados terminarem.
- Os computadores guardam arquivos em dispositivos de armazenamento secundários, como discos rígidos.

Seção 15.2 Arquivos e fluxos

- O Java vê cada arquivo como um fluxo sequencial de bytes.
- Cada sistema operacional fornece um mecanismo para determinar o final de um arquivo, como um marcador de fim de arquivo ou uma contagem dos bytes totais nele.
- Fluxos baseados em bytes representam dados no formato binário.
- Fluxos baseados em caracteres representam dados como sequências de caracteres.
- Arquivos criados usando fluxos baseados em bytes são arquivos binários. Arquivos criados usando fluxos baseados em caracteres são arquivos de texto. Os arquivos de texto podem ser lidos por editores de textos, enquanto arquivos binários são lidos por um programa que converte os dados em um formato legível para humanos.
- O Java também pode associar fluxos a diferentes dispositivos. Três objetos de fluxo são relacionados aos dispositivos quando um programa Java inicia a execução — `System.in`, `System.out` e `System.err`.

Seção 15.3 Usando classes e interfaces NIO para obter informações de arquivo e diretório

- Uma `Path` representa o local de um arquivo ou diretório. Objetos Path não abrem arquivos nem oferecem recursos de processamento de arquivo.
- A classe `Paths` é usada para obter um objeto `Path` representando um local de arquivo ou diretório.
- A classe `Files` fornece métodos `static` para manipulações comuns de arquivos e diretórios, incluindo métodos para copiar arquivos; criar e excluir arquivos e diretórios; obter informações sobre arquivos e diretórios; ler o conteúdo dos arquivos; obter objetos que permitem manipular o conteúdo dos arquivos e diretórios; etc.
- Um `DirectoryStream` permite que um programa itere pelo conteúdo de um diretório.
- O método `static get` da classe `Paths` converte uma `String` representando o local de um arquivo ou diretório em um objeto `Path`.
- A entrada e saída baseada em caracteres pode ser realizada com as classes `Scanner` e `Formatter`.
- A classe `Formatter` permite a saída de dados formatados para a tela ou para um arquivo de uma maneira semelhante a `System.out.printf`.
- Um caminho absoluto contém todos os diretórios desde o diretório raiz que levam a um arquivo ou diretório específico. Cada arquivo ou diretório em uma unidade de disco tem o mesmo diretório-raiz em seu caminho.
- Um caminho relativo normalmente inicia a partir do diretório em que o aplicativo começou a execução.
- O método `Files static exists` recebe um `Path` e determina se ele existe (como um arquivo ou como um diretório) no disco.
- O método `Path getFileName` obtém o nome de `String` de um arquivo ou diretório sem nenhuma informação sobre o local.
- O método `Files static isDirectory` recebe um `Path` e retorna um `boolean`, indicando se esse `Path` representa um diretório no disco.
- O método `Path isAbsolute` retorna um `boolean`, indicando se um `Path` representa um caminho absoluto para um arquivo ou diretório.
- O método `Files static getLastModifiedTime` recebe um `Path` e retorna um `FileTime` (pacote `java.nio.file.attribute`), indicando quando o arquivo foi modificado pela última vez.
- O método `Files static size` recebe um `Path` e retorna um `long`, representando o número de bytes no arquivo ou diretório. Para diretórios, o valor retornado é específico da plataforma.
- O método `Path toString` retorna uma representação `String` do `Path`.
- O método `Path toAbsolutePath` converte o `Path` no que é chamado de um caminho absoluto.
- O método `Files static newDirectoryStream` retorna um `DirectoryStream<Path>` contendo os objetos `Path` para o conteúdo de um diretório.
- Um caractere separador é utilizado para separar diretórios e arquivos no caminho.

Seção 15.4 Arquivos de texto de acesso sequencial

- O Java não impõe nenhuma estrutura a um arquivo. Você deve estruturar os arquivos para atender às necessidades do seu aplicativo.
- Para recuperar em sequência dados de um arquivo, os programas em geral começam a partir do início do arquivo e leem todos os dados consecutivamente até que as informações desejadas sejam encontradas.
- Os dados em muitos arquivos sequenciais não podem ser modificados sem o risco de destruir outros no arquivo. Registros em um arquivo de acesso sequencial geralmente são atualizados regravando todo o arquivo.

Seção 15.5 Serialização de objeto

- O Java fornece um mecanismo chamado serialização de objeto, que permite a objetos inteiros serem gravados ou lidos a partir de um fluxo.
- Um objeto serializado é representado como uma sequência de bytes que inclui os dados dele, bem como as informações sobre seu tipo e a natureza dos dados armazenados.
- Depois que um objeto serializado foi gravado em um arquivo, ele pode ser lido a partir desse arquivo e desserializado para recriar o objeto na memória.
- As classes `ObjectInputStream` e `ObjectOutputStream` permitem que objetos inteiros sejam lidos ou gravados em um fluxo (possivelmente um arquivo).
- Somente classes que implementam a interface `Serializable` podem ser serializadas e desserializadas.
- A interface `ObjectOutput` contém o método `writeObject`, que recebe um `Object` como um argumento e grava as informações em um `OutputStream`. Uma classe que implementa essa interface, como `ObjectOutputStream`, garantiria que o `Object` fosse `Serializable`.
- A interface `ObjectInput` contém o método `readObject`, que lê e retorna uma referência a um `Object` a partir de um `InputStream`. Depois que um objeto foi lido, podemos fazer uma coerção da sua referência para o tipo real do objeto.

Seção 15.6 Abrindo arquivos com `JFileChooser`

- A classe `JFileChooser` é utilizada para exibir um diálogo que permite aos usuários de um programa selecionar facilmente arquivos ou diretórios em uma GUI.

Seção 15.7 (Opcional) Classes `java.io` adicionais

- `InputStream` e `OutputStream` são classes `abstract` para realizar E/S baseada em bytes.
- Pipes são canais de comunicação sincronizados entre threads. Uma thread envia dados por meio de um `PipedOutputStream`. O thread alvo lê informações do pipe por um `PipedInputStream`.
- Um fluxo de filtro fornece funcionalidade adicional, como agregar bytes de dados em unidades de tipo primitivo significativas. `FilterInputStream` e `FilterOutputStream` são geralmente estendidos, assim algumas das capacidades de filtragem são fornecidas por suas subclasses concretas.
- Um `PrintStream` realiza a saída de texto. `System.out` e `System.err` são `PrintStreams`.
- A interface `DataInput` descreve os métodos para ler tipos primitivos a partir de um fluxo de entrada. As classes `DataInputStream` e `RandomAccessFile` implementam essa interface.
- A interface `DataOutput` descreve os métodos para gravar tipos primitivos em um fluxo de saída. As classes `DataOutputStream` e `RandomAccessFile` implementam, cada uma, essa interface.
- Armazenamento em buffer (*buffering*) é uma técnica de aprimoramento do desempenho de E/S. Ela reduz o número de operações de E/S combinando saídas menores na memória. O número de operações físicas de E/S é muito menor do que o de solicitações de E/S emitidas pelo programa.
- Com um `BufferedOutputStream`, cada operação de saída é direcionada para um buffer grande o suficiente para conter os dados de muitas operações de saída. A transferência para o dispositivo de saída é realizada em uma grande operação de saída física quando o buffer está cheio. Um buffer parcialmente preenchido pode ser forçado para o dispositivo a qualquer momento, invocando o método `flush` do objeto de fluxo.
- Com um `BufferedInputStream`, muitos fragmentos ou trechos "lógicos" de dados de um arquivo são lidos como uma grande operação física de entrada em um buffer de memória. Quando um programa solicita dados, eles são tirados do buffer. Quando o buffer está vazio, a próxima operação de entrada física real é executada.
- Um `ByteArrayInputStream` lê a partir de um array de `byte` na memória. Um `ByteArrayOutputStream` envia uma saída para um array de `byte` na memória.
- Um `SequenceInputStream` concatena vários `InputStreams`. Quando o programa alcança o final de um fluxo de entrada, este se fecha e o próximo fluxo na sequência se abre.
- As classes `Reader` e `Writer` `abstract` são fluxos baseados em caracteres Unicode. A maioria dos fluxos baseados em bytes tem classes `Reader` ou `Writer` concretas correspondentes baseadas em caracteres.
- As classes `BufferedReader` e `BufferedWriter` armazenam em buffer os fluxos baseados em caracteres.
- As classes `CharArrayReader` e `CharArrayWriter` manipulam arrays `char`.
- Um `LineNumberReader` é um fluxo de caracteres armazenado em buffer que rastreia o número de linhas lidas.
- As classes `FileReader` e `FileWriter` realizam E/S de arquivos baseados em caracteres.
- A classe `PipedReader` e a `PipedWriter` implementam fluxos de caracteres redirecionados (*piped*) para transferência de dados entre threads.
- As classes `StringReader` e `StringWriter` leem e gravam caracteres em `Strings`, respectivamente. Um `PrintWriter` grava caracteres em um fluxo.

Exercícios de revisão

15.1 Determine se cada uma das sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Você deve criar explicitamente os objetos de fluxo `System.in`, `System.out` e `System.err`.

b) Ao ler dados de um arquivo utilizando a classe `Scanner`, se você quiser fazer isso múltiplas vezes, o arquivo deve ser fechado e reaberto para ler a partir do início dele.

c) O método `Files` `static` `exists` recebe um `Path` e determina se ele existe (como um arquivo ou um diretório) no disco.

d) Arquivos binários são legíveis em um editor de texto.

e) Um caminho absoluto contém todos os diretórios, desde o diretório-raiz, que levam a um arquivo ou diretório específico.

f) A classe `Formatter` contém o método `printf`, que permite gerar a saída de dados formatados para a tela ou para um arquivo.

15.2 Cumpra as seguintes tarefas, supondo que cada uma se aplica ao mesmo programa:

a) Escreva uma instrução que abre o arquivo `"oldmast.txt"` para entrada — utilize a variável `Scanner` `inOldMaster`.

b) Escreva uma instrução que abre o arquivo `"trans.txt"` para entrada — utilize a variável `Scanner` `inTransaction`.

c) Escreva uma instrução que abre arquivo `"newmast.txt"` para saída (e criação) — utilize a variável `formatter` `outNewMaster`.

d) Escreva as instruções necessárias para ler um registro do arquivo `"oldmast.txt"`. Use os dados para criar um objeto da classe `Account` — utilize a variável `Scanner` `inOldMaster`. Suponha que essa classe `Account` é idêntica àquela na Figura 15.9.

e) Escreva as instruções necessárias para ler um registro do arquivo `"trans.txt"`. O registro é um objeto da classe `TransactionRecord` — utilize a variável `Scanner` `inTransaction`. Suponha que essa classe `TransactionRecord` contenha o método `setAccount` (que recebe um `int`) para configurar o número de conta e o método `setAmount` (que recebe um `double`) a fim de estabelecer o valor monetário da transação.

f) Escreva uma instrução que gera a saída de um registro para o arquivo `"newmast.txt"`. O registro é um objeto do tipo `Account` — utilize a variável `Formatter outNewMaster`.

15.3 Realize as seguintes tarefas, supondo que cada uma se aplica ao mesmo programa:

a) Escreva uma instrução que abre o arquivo `"oldmast.ser"` para entrada — utilize a variável `ObjectInputStream` para empacotar um objeto `InputStream`.

b) Escreva uma instrução que abre o arquivo `"trans.ser"` para entrada — utilize a variável `ObjectInputStream` para empacotar um objeto `InputStream`.

c) Escreva uma instrução que abre arquivo `"newmast.ser"` para saída (e criação) — utilize a variável `ObjectOutputStream outNewMaster` para empacotar um `OutputStream`.

d) Escreva uma instrução que lê um registro no arquivo `"oldmast.ser"`. O registro é um objeto da classe `Account` — utilize a variável `ObjectInputStream inOldMaster`. Assuma que essa classe `Account` é a mesma que aquela na Figura 15.9.

e) Escreva uma instrução que lê um registro no arquivo `"trans.ser"`. O registro é um objeto da classe `TransactionRecord` — utilize a variável `ObjectInputStream inTransaction`.

f) Escreva uma instrução que gera a saída de um registro do tipo `Account` para o arquivo `"newmast.ser"` — use a variável `ObjectOutputStream outNewMaster`.

Respostas dos exercícios de revisão

15.1 a) Falsa. Esses três fluxos são criados para você quando um aplicativo Java começa a executar. b) Verdadeira. c) Verdadeira. d) Falsa. Arquivos de texto são legíveis para seres humanos em um editor de texto. Arquivos binários podem ser legíveis para seres humanos, mas apenas se os bytes no arquivo representam os caracteres ASCII. e) Verdadeira. f) Falsa. A classe `Formatter` contém o método `format`, que permite gerar a saída de dados formatados para a tela ou para um arquivo.

15.2 a)
```
Scanner inOldMaster = new Scanner(Paths.get("oldmast.txt"));
```
b)
```
Scanner inTransaction = new Scanner(Paths.get("trans.txt"));
```
c)
```
Formatter outNewMaster = new Formatter("newmast.txt");
```
d)
```
Account account = new Account();
account.setAccount(inOldMaster.nextInt());
account.setFirstName(inOldMaster.next());
account.setLastName(inOldMaster.next());
account.setBalance(inOldMaster.nextDouble());
```
e)
```
TransactionRecord transaction = new Transaction();
transaction.setAccount(inTransaction.nextInt());
transaction.setAmount(inTransaction.nextDouble());
```
f)
```
outNewMaster.format("%d %s %s %.2f%n",
   account.getAccount(), account.getFirstName(),
   account.getLastName(), account.getBalance());
```

15.3 a)
```
ObjectInputStream inOldMaster = new ObjectInputStream(
      Files.newInputStream(Paths.get("oldmast.ser")));
```
b)
```
ObjectInputStream inTransaction = new ObjectInputStream(
   Files.newOutputStream(Paths.get("trans.ser")));
```
c)
```
ObjectOutputStream outNewMaster = new ObjectOutputStream(
   Files.newOutputStream(Paths.get("newmast.ser")));
```
d)
```
Account = (Account) inOldMaster.readObject();
```
e)
```
transactionRecord = (TransactionRecord) inTransaction.readObject();
```
f)
```
outNewMaster.writeObject(newAccount);
```

Questões

15.4 (***Correspondência de arquivos***) O Exercício de revisão 15.2 pede que você escreva uma série de instruções únicas. De fato, essas instruções formam o núcleo de um importante tipo de programa processador de arquivo, a saber, um programa de correspondência de arquivo (*file-matching program*). Em processamento de dados comercial, é comum ter vários arquivos em cada sistema de aplicativo. Em um sistema de contas a receber, por exemplo, há em geral um arquivo mestre contendo informações detalhadas sobre cada cliente, como seu nome, endereço, número de telefone, saldo, limite de crédito, termos de desconto, arranjos de contrato e possivelmente um histórico condensado de compras recentes e pagamentos em dinheiro.

À medida que as transações ocorrem (isto é, vendas são feitas e pagamentos chegam pelo correio), as informações sobre elas são inseridas em um arquivo. No fim de cada período de negócios (um mês para algumas empresas, uma semana para outras e um dia em alguns casos), o arquivo de transações (chamado `"trans.txt"`) é aplicado ao arquivo-mestre (chamado `"oldmast.txt"`) para atualizar o registro de compras e pagamentos de cada conta. Durante uma atualização, o arquivo-mestre é regravado como o arquivo `"newmast.txt"`, que é então utilizado no fim do período seguinte de negócios para começar o processo de atualização novamente.

Programas de correspondência de arquivo devem lidar com certos problemas que não surgem em programas de um único arquivo. Por exemplo, nem sempre ocorre uma correspondência. Se um cliente no arquivo-mestre não fez nenhuma compra ou pagamento em dinheiro no período de negócios atual, nenhum registro para esse cliente aparecerá no arquivo de transações. De maneira semelhante, um cliente que fez alguma compra ou pagamento em dinheiro talvez tenha mudado recentemente para essa comunidade e, se foi o caso, a empresa pode não ter tido uma oportunidade de criar um registro-mestre para ele.

Escreva um programa completo de correspondência de arquivos de contas a receber. Utilize o número de conta em cada arquivo como a chave de registro para propósitos de correspondência. Assuma que cada arquivo é um arquivo de texto sequencial com registros armazenados em ordem de número de conta crescente.

a) Defina a classe `TransactionRecord`. Os objetos dessa classe contêm um número de conta e valor monetário para a transação. Forneça métodos para modificar e recuperar esses valores.

b) Modifique a classe `Account` na Figura 15.9 para incluir o método `combine`, que recebe um objeto `TransactionRecord` e combina o saldo de `Account` e o valor monetário de `TransactionRecord`.

c) Escreva um programa para criar dados a fim de testar o programa. Utilize os dados de exemplo da conta nas figuras 15.14 e 15.15. Execute o programa para criar os arquivos `trans.txt` e `oldmast.txt` a serem utilizados por seu programa de correspondência de arquivos.

Número de conta do arquivo-mestre	Nome	Saldo
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

Figura 15.14 | Dados de exemplo para o arquivo-mestre.

Número de conta do arquivo de transação	Quantia da transação
100	27.14
300	62.11
400	100.56
900	82.17

Figura 15.15 | Dados de exemplo para o arquivo de transações.

d) Crie a classe `FileMatch` para realizar a funcionalidade de correspondência de arquivos. A classe deve conter métodos que leem `oldmast.txt` e `trans.txt`. Quando uma correspondência ocorre (isto é, registros com o mesmo número de conta aparecem tanto no arquivo-mestre como no arquivo de transações), adicione o valor monetário no registro de transação ao saldo atual no registro-mestre e grave o registro em `"newmast.txt"`. (Suponha que compras sejam indicadas por montantes positivos no arquivo de transações, e os pagamentos, por valores monetários negativos.) Caso haja um registro-mestre para uma conta particular, mas nenhum registro de transação correspondente, simplesmente grave o registro-mestre em `"newmast.txt"`. Se houver um registro de transação, mas nenhum registro-mestre correspondente, imprima a mensagem `"Unmatched transaction record for account number..."` `[Registro de transação não correspondente para o número da conta]` em um arquivo de log (preencha o número da conta a partir do registro de transação). O arquivo de log deve ser um arquivo de texto chamado `"log.txt"`.

15.5 (***Correspondência de arquivos com múltiplas transações***) É possível (e, na verdade, comum) ter vários registros de transações com a mesma chave de registro. Essa situação ocorre, por exemplo, quando um cliente faz várias compras e pagamentos em dinheiro durante um período de negócios. Reescreva seu programa de correspondência de arquivo de contas a receber a partir da Questão 15.4 para prever a possibilidade de lidar com vários registros de transações com a mesma chave de registro. Modifique os dados de teste de `CreateData.java` a fim de incluir registros de transações adicionais na Figura 15.16.

Número de conta	Quantia em dólar
300	83.89
700	80.78
700	1.53

Figura 15.16 | Registros de transações adicionais.

15.6 (***Correspondência de arquivos com serialização de objetos***) Recrie sua solução para a Questão 15.5 utilizando a serialização de objetos. Utilize as instruções do Exercício 15.3 como sua base para esse programa. Talvez você queira criar aplicativos a fim de ler os dados armazenados nos arquivos `.ser` — o código na Seção 15.5.2 pode ser modificado para esse propósito.

15.7 (***Gerador de palavra de número de telefone***) Os teclados de telefone padrão contêm os dígitos de zero a nove. Os números 2 a 9 têm três letras associadas a cada um (Figura 15.17). Muitas pessoas acham difícil memorizar números de telefone, então utilizam a correspondência entre dígitos e letras para criar palavras de sete letras que correspondem a seus números de telefone. Por exemplo, uma pessoa cujo número de telefone é 686-2377 talvez adote a correspondência indicada na Figura 15.17 para desenvolver a palavra de sete letras "NUMBERS". Cada palavra de sete letras se associa a exatamente um número de telefone de sete dígitos. Um restaurante que deseja ampliar seu esquema de entregas em domicílio ("takeout", em inglês) seguramente poderia fazer isso com o número 825-3688 (isto é, "TAKEOUT").

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 R S		

Figura 15.17 | Dígitos e letras do teclado do telefone.

Cada número de telefone de sete letras corresponde a diversas palavras de sete letras, mas a maioria delas representa justaposições irreconhecíveis das letras. É possível, porém, que o proprietário de uma barbearia ficasse contente em saber que o número de telefone de seu salão, 424-7288, corresponde a "HAIRCUT" (que significa "corte de cabelo"). Um veterinário com o número de telefone 738-2273 ficaria satisfeito em saber que seu número corresponde à palavra de sete letras "PETCARE" (que significa "assistência a animais de estimação"). Um vendedor de automóveis ficaria animado ao saber que o número de telefone de sua loja, 639-2277, corresponde a "NEWCARS" (que significa "carros novos").

Escreva um programa que, dado um número de sete dígitos, utiliza um objeto `PrintStream` para gravar em um arquivo cada possível combinação de palavras de sete letras correspondente a esse número. Há 2.187 (3^7) dessas combinações. Evite números de telefone com os dígitos 0 e 1.

15.8 (***Pesquisa entre alunos***) A Figura 7.8 contém um array de respostas a uma pesquisa que é codificado diretamente no programa. Suponha que queremos processar os resultados dessa pesquisa que são armazenados em um arquivo. Este exercício requer dois programas separados. Primeiro, crie um aplicativo que solicita ao usuário respostas à pesquisa e gera a saída de cada resposta para um arquivo. Utilize um `Formatter` para criar um arquivo chamado `numbers.txt`. Cada inteiro deve ser escrito com o método `format`. Então modifique o programa que aparece na Figura 7.8 para ler as respostas à pesquisa a partir de `numbers.txt`. As respostas devem ser lidas do arquivo utilizando um `Scanner`. Use o método `nextInt` para inserir um número inteiro de cada vez a partir do arquivo. O programa precisa continuar a ler respostas até alcançar o fim desse arquivo. A saída dos resultados deve ser gerada no arquivo de texto `"output.txt"`.

15.9 (***Adicionando serialização de objetos ao aplicativo de desenho MyShape***) Modifique a Questão 12.17 para permitir ao usuário salvar um desenho em um arquivo ou carregar uma produção anterior usando a serialização de objetos. Adicione botões **Load** (para ler objetos de um arquivo) e **Save** (para gravar objetos em um arquivo). Utilize um `ObjectOutputStream` a fim de gravar no arquivo e um `ObjectInputStream` para ler o arquivo. Escreva o array de objetos `MyShape` utilizando o método `writeObject` (classe `ObjectOutputStream`) e leia o array utilizando o método `readObject` (`ObjectInputStream`). O mecanismo de serialização de objeto pode ler ou gravar arrays inteiros — não é necessário manipular cada elemento do array de objetos `MyShape` individualmente. Simplesmente é exigido que todas as formas sejam `Serializable`. Para os dois botões **Load** e **Save**, use um `JFileChooser` para permitir que o usuário selecione o arquivo em que as formas serão armazenadas ou do qual elas serão lidas. Quando o usuário executa o programa pela primeira vez, nenhuma forma deve aparecer na tela. O usuário pode exibir formas abrindo um arquivo salvo anteriormente ou desenhando novas formas. Uma vez que há formas na tela, os usuários podem salvá-las para um arquivo usando o botão **Save**.

Fazendo a diferença

15.10 (***Scanner de phishing***) *Phishing* é uma forma de roubo de identidade pela qual, em um e-mail, um remetente fingindo ser uma fonte confiável tenta adquirir informações privadas, como nomes de usuário, senhas, números de cartões de crédito e número de previdência social. E-mails contendo *phishing* fingindo ser de bancos populares, empresas de cartões de crédito, sites de leilão, redes sociais e serviços de pagamento on-line podem parecer bem legítimos. Essas mensagens fraudulentas geralmente fornecem links para sites falsos nos quais você é solicitado a inserir informações sigilosas.

Faça uma pesquisa on-line sobre golpes de phishing. Verifique também o Anti-Phishing Working Group (<www.antiphishing.org>) e o site Cyber Investigations do FBI (<www.fbi.gov/about-us/investigate/cyber/cyber>), nos quais você encontrará informações sobre os golpes mais recentes e como se proteger.

Crie uma lista de 30 palavras, frases e nomes de empresas comumente encontrados em mensagens de *phishing*. Atribua um ponto a cada uma com base na sua estimativa da probabilidade de estar em uma mensagem desse gênero (por exemplo, um ponto se é pouco provável, dois pontos se moderadamente provável ou três pontos se altamente provável). Elabore um aplicativo que verifica em um arquivo de texto esses termos e frases. Para cada ocorrência de uma palavra-chave ou frase no arquivo de texto, some o ponto atribuído aos totais para essa palavra ou frase. A cada palavra-chave ou frase, gere uma linha com elas, o número de ocorrências e os pontos totais. Então, mostre os pontos totais para a mensagem inteira. Seu programa atribui um total de pontos altos a alguns dos e-mails de *phishing* reais que você recebeu? Ele atribui uma total de pontos altos a alguns e-mails legítimos que você recebeu?