Capítulo 10

Arquivos e Streams

Dados são representados em computadores como seqüências de bytes. Embora se possa criar programas onde os dados são todos transientes, isto é, a sua representação como seqüência de bytes só é mantida durante a execução do programa, grande parte das aplicações não pode se limitar ao armazenamento de dados apenas em memória. Em outras palavras, estas aplicações não podem ter seus dados perdidos quando são finalizadas, exigindo uma forma de armazenamento mais sofisticada que permita sua recuperação futura. Nestes casos, a representação dos dados deve persistir entre as várias execuções do programa. Para isto, podemos armazenar estes conjuntos de dados na memória secundária de um sistema, usualmente unidades de disco rígido ou flexível, CD-ROM, DVD ou fitas magnéticas.

Este capítulo irá tratar de como obter uma seqüência de bytes de uma fonte de dados, e de como enviar seqüências de bytes a qualquer destino que possa receber estes dados. Estes destinos e estas fontes de dados podem ser – e na maioria das vezes são – arquivos, mas também podem ser conexões de rede e até mesmo blocos de memória. Tenha em mente, portanto, que informações armazenadas em arquivos e informações obtidas de uma conexão de rede são essencialmente tratadas da mesma forma pela linguagem Java.

10.1 - Notação Multiplataforma para Trilhas em Sistemas de Arquivos

Antes de nos aprofundarmos no mecanismo fornecido por Java para lidar com seqüências de bytes, apresentamos como Java trata as diferentes notações usadas pelos diversos sistemas operacionais para a especificação de trilhas de diretórios e arquivos.

Diferentes sistemas operacionais, possuem diferentes formas de denotar trilhas de diretórios e arquivos. Por exemplo, o estilo Windows requer a especificação do drive a partir de onde a trilha parte e o uso da contra-barra ("\") para separar os diretórios da trilha. Sistemas UNIX, por sua vez, utilizam a barra ("/") como separador e não requerem a especificação do drive.

Para poder lidar com a demanda de programas multiplataforma, Java necessitou criar uma representação abstrata para as trilhas de arquivos e diretórios. A classe `File` oferece uma perspectiva abstrata e independente de plataforma sobre trilhas hierárquicas de diretórios. Uma trilha abstrata é composta por duas componentes:

1. uma string opcional representando um prefixo, tais como, um especificador de drive, "/" para a raiz dos diretórios UNIX ou uma "\\" para o Windows (uma vez que a contra barra ("\") é o caractere de escape na linguagem Java, precisa-se usar \\ para as trilhas no estilo do Windows ("C:\\Windows\\win.ini")).
2. Uma seqüência de zero ou mais de strings com nomes dos diretórios da trilha (em tempo, o último nome da seqüência pode denotar um arquivo).

A conversão de uma trilha do sistema para (ou a partir de) uma trilha abstrata é inerentemente dependente da plataforma. Quando uma trilha abstrata é convertida para a trilha do sistema, cada nome é separado do seguinte através de um caracter separador, definido no atributo estático `separator` da classe `File`. Quando a situação é inversa, a trilha do sistema é separada para se obter as strings que compõem a trilha abstrata.

Além de representar trilhas abstratas, a classe `File` fornece métodos independentes da plataforma para listar diretórios, obter atributos de arquivos e para renomear e excluir arquivos e diretórios, tais como, `getName()`, `getPath()`, `canWrite()`, `isDirectory()` e outros.. É importante destacar que em Java um diretório é apenas um outro arquivo. Você pode criar um objeto da classe `File` para representar um diretório e usá-lo para acessar outros arquivos. Os exemplos seguintes mostram como criar objetos da classe `File`:

```
File meuArquivo1 = new File("meuArq1.txt");
File meuArquivo2 =
     new File("c:\\meuDir", "meuArq2.txt");
File meuDir = new File("c:\\meuDir");
File meuArquivo3 = new File(meuDir, "meuArq3.txt");
```

A primeira declaração cria uma instância da classe `File` e a associa a trilha abstrata composta pelo diretório corrente e pelo nome `"meuArq1.txt"`. A segunda declaração associa a instância criada de `File` a trilha abstrata composta pelo diretório `"c:\meuDir"` e pelo nome `"meuArq2.txt"`. A terceira declaração associa a instância a trilha abstrata composta pelo diretório `"c:\meuDir"`. A quarta declaração faz o mesmo que a segunda, só que associando o nome `"meuArq3.txt"`.

Observe que você deve criar um objeto para cada arquivo que será manipulado. A escolha do construtor a ser usado depende dos outros arquivos que você vai acessar. Se você só usará um arquivo na sua aplicação, deve usar o primeiro ou segundo construtor. Contudo, se você usará vários arquivos de um diretório comum, pode ser mais fácil utilizar a combinação do terceiro com o quarto construtor.

10.2 – Streams

Em Java, *streams* são seqüência de bytes. Objetos nos quais podemos escrever uma seqüência de bytes são denominadas input streams. Já os objetos dos quais podemos ler seqüências de bytes são denominadas output streams. Elas são definidas através das classes abstratas `InputStream` e `OutputStream`. Uma vez que streams orientadas a bytes são inconvenientes para o processamento de informações armazenadas no padrão Unicode (lembre-se que o Unicode usa 2 bytes para representar cada caractere), existe uma hierarquia de classes a parte para a manipulação deste tipo de arquivos, que herdam das superclasses abstratas `Reader` e `Writer`. Estas classes possuem operações de escrita e leitura baseadas nos 2 bytes utilizados pelo padrão Unicode para a representação de caracteres.

Lembre-se que, no capítulo 6, você aprendeu que as classes abstratas permitem a criação de um ancestral comum para um conjunto de classes, o que lhes possibilita o uso do polimorfismo. Você verá ainda neste capítulo que as diversas classes concretas existentes na linguagem para a manipulação de streams derivam das 4 classes abstratas mencionadas acima.

10.2.1 – Lendo e Escrevendo Bytes

A classe InputStream possui o seguinte método abstrato:

```
public abstract int read() throws IOException {
```

Este método lê um byte e retorna o byte lido, ou retorna –1 caso seja encontrado o final da fonte de dados. As classes concretas que implementam a classe `InputStream`, possuem seus métodos abstratos sobrecarregados. Por exemplo, na classe `FileInputStream`, o método `read` lê e retorna um byte de um arquivo. `System.in` é um objeto pré-definido de uma subclasse de `InputStream` que nos permite a leitura de dados do teclado do computador.

A classe `InputStream` também possui métodos não abstratos para a leitura de um array de bytes ou para ignorar um número de bytes. Estes métodos concretos fazem uso do método abstrato `read`, portanto as subclasses de `InputStream` necessitam implementar apenas um método abstrato.

Similarmente, a classe `OutputStream` define o método abstrato:

```
public abstract void write(int b) throws IOException
```

Que escreve um byte em uma output stream.

Ambos os métodos `read` e `write` podem bloquear um processo até que o byte tenha sido completamente lido/escrito. Isto significa que, caso a stream não possa ser imediatamente lida ou escrita (por exemplo por uma conexão de rede ocupada), a linguagem Java suspende o processo responsável por esta chamada. Isto permite que outros processos possam ser executados enquanto aquele processo aguarda a liberação da stream para poder retornar a sua execução.

O método `available` permite que você cheque o número de bytes que estão atualmente disponíveis para leitura. Isto significa que um processo contendo um fragmento de código como o exemplo abaixo dificilmente será bloqueado:

```
int bytesAvailable = System.in.available();
if (bytesAvailable > 0) {
      byte[] data = new byte[bytesAvailable];
      System.in.Read(data);
}
```

Quando você tiver terminado de ler ou escrever em uma stream, é importante que você a feche, usando o método `close`. Pelo fato das streams serem recursos do sistema operacional, é desejável que uma aplicação aloque em determinado momento, apenas aqueles recursos dos quais realmente faz uso. Caso contrário, esta aplicação poderá comprometer o desempenho do sistema. Lembre-se que o seu processo provavelmente não será o único processo em execução no sistema.

Pode-se notar que a maior parte dos programadores da linguagem Java raramente fazem uso das classes que implementam os métodos `read` e `write`, simplesmente pelo fato de dificilmente necessitarem escrever ou ler bytes em streams.

Os tipos de dados que provavelmente você se interessará em armazenar conterão números, strings e objetos. A linguagem Java oferece diversas classes streams derivadas das classes básicas `InputStream` e `OutputStream` que lhe permitirão manipular estes dados de mais alto nível.

10.3 – A Hierarquia de Streams

Diferentemente da linguagem C, que possui apenas um tipo de arquivo, ou da linguagem Visual Basic, que possui três tipos de arquivos, a linguagem Java possui um conjunto com

mais de 60 tipos de diferentes streams. Os projetistas da linguagem acreditavam que esta diversidade de streams permitiria que os usuários as usassem de forma mais adequada, o que diminuiria substancialmente o número de erros de programação, muito comuns durante a manipulação de arquivos. Mas o que acaba acontecendo na prática é que a complexidade das bibliotecas de streams acaba por intimidar os programadores.

Vamos mostrar agora a hierarquia destas streams. Quatro classes abstratas formam a base da hierarquia: `InputStream`, `OutputStream`, `Reader` e `Writer`. Por serem classes abstratas, não podemos criar instâncias (objetos) destas classes, mas nada nos impede de fazer uso do polimorfismo. Por exemplo, a classe `URL` possui o método `openStream` que retorna um objeto na forma `InputStream`. Isto não significa que este objeto seja uma instância desta classe, e sim de uma classe concreta que implemente a classe abstrata `InputStream`. Como foi mencionado anteriormente, as classes `InputStream` e `OutputStream` permitem a leitura/escrita de bytes ou arrays de bytes; estas classes não possuem métodos para leitura/escrita de números ou strings. Para realizar esta tarefa, você precisará usar classes mais específicas, como a `DataInputStream` ou `DataOutputStream`, que são capazes de escrever ou ler todos os tipos básicos da linguagem.

Por outro lado, para manipular textos no formato Unicode, você usará subclasses das streams `Reader` e `Writer`. Os métodos básicos destas classes são similares aos métodos das classes `InputStream` e `OutputStream`.

```
public abstract int read() throws IOException
public abstract void write(int b) throws IOException
```

Estes métodos funcionam exatamente como os métodos `read` e `write` das classes `InputStream` e `OutputStream`, exceto pelo fato de retornarem um caractere Unicode (representado por um inteiro de 0 a 65535), ou –1 caso seja encontrado o fim do arquivo.

10.3.1 – Filtros Para Streams Em Camadas

As streams `FileInputStream` e `FileOutputStream` oferecem serviços de leitura e escrita em arquivos em disco. Seus métodos construtores recebem a localização completa do arquivo (trilha) ou simplesmente o nome do arquivo (neste caso, o diretório usado pela linguagem será o diretório corrente).

```
FileInputStream fin =
     new FileInputStream("empregado.dat");
```

O código acima procura pelo arquivo chamado `empregado.dat` no diretório corrente (geralmente o diretório em que a aplicação em execução está localizada).

Como as classes abstratas `InputStream` e `OutputStream`, estas classes (`FileInputStream` e `FileOutputStream`) suportam apenas leitura e escrita de bytes em arquivos. Como você viu anteriormente, podemos usar a `DataInputStream` para ler tipos numéricos ou strings:

```
DataInputStream din = ...;
double = din.readDouble();
```

Contudo, ao mesmo tempo em que a stream `FileInputStream` não possui métodos para ler tipos numéricos, a stream `DataInputStream` não possui métodos para obter dados de um arquivo.

Java possui um artifício para separar dois tipos de responsabilidades. Algumas streams (como `FileInputStream` e a stream retornada pelo método `openStream` da classe `URL`) podem obter bytes de arquivos ou locais mais exóticos (conexões de rede ou da própria memória RAM, por exemplo). Outras streams (como a `DataInputStream` e a `PrintWriter`) podem organizar bytes em tipos de dados mais apropriados. O programador é responsável por combinar estas duas funcionalidades, passando uma destas streams para o construtor da outra stream. Por exemplo, primeiramente crie uma `FileInputStream` e depois passe-a como argumento no construtor da `DataInputStream`.

```
FileInputStream fin =
     new FileInputStream("empregado.dat");
DataInputStream din =
     new DataInputStream(fin);
double s = din.readDouble();
```

É importante deixar claro que a nova stream criada no código acima não corresponde a um novo arquivo em disco. A nova stream continua acessando os dados do arquivo ligado à `FileInputStream`.

Ao olhar as figuras 10.1 e 10.3, você verá as classes `FilterInputStream` e `FilterOutputStream`. Você pode combinar suas subclasses para conseguir uma nova stream com a funcionalidade desejada. Por exemplo, o padrão é que as streams não sejam bufferizadas. O que significa que cada chamada a `read` entra em contato com o sistema operacional para requerer a leitura de mais um byte. Se você quiser bufferizar a leitura dos dados do arquivo empregado.dat, localizado no diretório corrente, você deverá fazer uso da longa seqüência de construtores abaixo:

```
DataInputStream din =
     new DataInputStream(
          new BufferedInputStream (
               new FileInputStream("empregado.dat")));
```

Note que colocamos a `DataInputStream` por último na cadeia de construtores devido ao fato de querermos usar seus métodos, e gostaríamos que estes métodos fizessem uso do método `read` bufferizado. Apesar do código acima não ser dos mais bonitos, faz-se necessária a combinação de streams para que a funcionalidade desejada seja atingida.

10.3.2 – A Família java.io.InputStream

A classe `java.io.InputStream` é uma classe abstrata significando que não pode ser usada diretamente. Ao invés disso dá origem a várias classes especializadas distintas cujo uso típico é em camadas associadas para prover a funcionalidade desejada, tal como proposto por um padrão de projeto onde objetos são utilizados em camadas para adicionar novas características a objetos mais simples de forma dinâmica e transparente, sendo que todos os objetos possuem a mesma interface. Embora um pouco mais complexo, este esquema oferece extrema flexibilidade na programação.

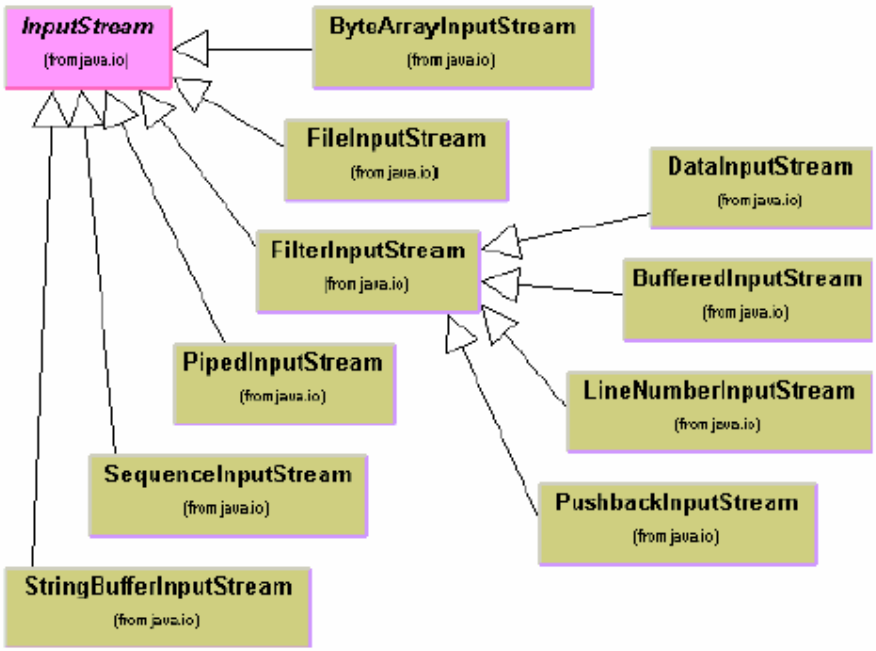

Figura 10. 1 – A Hierarquia InputStream

Os tipos de `InputStream`, descritos na Tabela 10.1, permitem o acesso a diferentes estruturas de dados, onde cada classe especializada oferece tratamento para um destes tipos. Todas as classes derivadas de `InputStream` são apropriadas para tratamento de arquivos orientados a bytes e em formato ASCII comum.

A classe `java.io.ByteArrayInputStream` permite que um buffer existente em memória seja usado como uma fonte de dados. Seu construtor recebe o buffer e pode ser usado em conjunto com a classe `java.io.FilterInputStream` para prover uma interface útil.

De forma semelhante a classe `java.io.StringBuffer` utiliza um objeto `StringBuffer` como argumento do construtor para transformar uma string numa fonte de dados cuja interface é dada pela associação com a classe `java.io.FilterInputStream`.

Para leitura de dados de arquivos utilizamos a classe `java.io.FileInputStream`, cujo construtor recebe um objeto `String`, `File` ou `FileDescriptor` contendo informações sobre o arquivo a ser aberto. Outra vez a associação com um objeto da classe `java.io.FilterInputStream` permite o acesso a uma interface útil para programação.

A classe `java.io.PipedInputStream` é utilizada em conjunto com a classe `java.io.PipedOutputStream` para a implementação de *pipes* (dutos de dados). Finalmente a classe `java.io.SequenceInputStream` fornece meios para concentrar a entrada de vários objetos `InputStream` numa única stream.

Tipo	Descrição
Vetor de Bytes	Dados são lidos de um vetor comum de bytes.
String	Dados são lidos de um objeto **StringBuffer**.
Arquivo	Dados são lidos de um arquivo existente no sistema de arquivos.
Pipe	Dados são obtidos de um *pipe*, facilidade oferecida pelo sistema operacional para redirecionamento de dados.
Sequência de Streams	Dados obtidos de diversas *streams* podem ser agrupados em uma única *stream*.
Outra fontes	Dados pode ser obtidos de outras fontes, tais como conexões com outros computadores através da rede.

Tabela 10. 1 – Tipos de InputStream

Através do uso de um `InputStream` associado a um `FilterInputStream` obtém-se formas mais especializadas de leitura:

Tipo	Descrição
DataInputStream	Oferece suporte para leitura direta de int, char, long etc.
BufferedInputStream	Efetua a leitura de dados através de um *buffer*, aumentando a eficiência das operações de leitura.
LineNumberInputStream	Mantêm controle sobre o número da linha lida na entrada.
PushbackInputStream	Permite a devolução do último caractere lido da *stream*.

Tabela 10.2 – Tipos de InputStream

As classes `DataInputStream` e `BufferedInputStream` são as classes mais versáteis desta família pois respectivamente permitem a leitura direta de tipos primitivos da linguagem e a leitura "bufferizada" de dados. Uma construção de classes em camadas típica é:

```
//UsaInputStream.java
import java.io.*;
public class UsaInputStream {
  public static void main(String [] args) {
    try {
     // constroi objeto stream para leitura de arquivo
     DataInputStream in =
        new DataInputStream (
          new BufferedInputStream (
            new FileInputStream("NomeArquivo.ext")));
     // string auxiliar para linha lida
     String linha;
     // string para armazenar todo texto lido
     String buffer = new String();
     // le primeira linha
     linha = in.readLine();
     // enquanto leitura não nula
     while (linha != null) {
       // adiciona linha ao buffer
       buffer += linha + "\n";
       // le proxima linha
```

```
        linha = in.readLine();
      }
      in.close(); // fecha stream de leitura
    } catch (IOException exc) {
      System.out.println("Erro de IO");
      exc.printStackTrace();
    }
}
```

Exemplo 10. 1 - Uso de InputStream

A classe `java.io.FilterInputStream` possui os seguintes construtores e métodos de interesse:

Método	Descrição
FilterInputStream(InputStream)	Constrói um objeto tipo **Filter** sobre o objeto **InputStream** especificado.
available()	Determina a quantidade de bytes disponíveis para leitura sem bloqueio.
close()	Fecha a *stream* e libera os recursos associados do sistema.
mark(int)	Marca a posição atual da *stream*. O valor fornecido indica a validade da marca para leituras adicionais.
markSupported()	Verifica se as operações **mark** e **reset** são suportadas.
read()	Lê o próximo byte disponível.
read(byte[])	Preenche o vetor de bytes fornecido como argumento.
reset()	Reposiciona a *stream* na última posição determinada por **mark**.
skip(long)	Descarta a quantidade de bytes especificada.

Tabela 10.3 – Métodos da Classe java.io.FilterInputStream

ATENÇÃO: O método `readLine` utilizado no exemplo pertence a classe `java.io.DataInputStream`. A partir da versão 1.1 do jdk, este método foi depreciado e não deve ser mais usado (ele não converte propriamente bytes em caracteres).

10.3.3 – A Família java.io.Reader

Tal qual a classe `java.io.InputStream`, a `java.io.Reader` é uma classe abstrata destinada a oferecer uma infra-estrutura comum para operações de leitura de streams orientadas a caractere (byte), embora suportando o padrão Unicode. Todas as suas subclasses implementam os métodos `read` e `close`, no entanto o fazem de forma a prover maior eficiência ou funcionalidade adicional através de outros métodos. Na Figura 10.2 temos ilustrada a hierarquia de classes baseadas na classe `java.io.Reader`.

A classe `java.io.BufferedReader` oferece leitura eficiente de arquivos comuns quando implementada em associação a classe `java.io.FileReader`, pois evita que operações simples de leitura sejam realizadas uma a uma, lendo uma quantidade maior de dados que a solicitada e armazenado tais dados num buffer, reduzindo o número de operações físicas. A classe `java.io.BufferedReader` serve de base para a classe `java.io.LineNumberReader`.

A classe `java.io.PipedReader` é utilizada em conjunto com `java.io.PipedWriter` para implementação de pipes. A classe `java.io.StringReader` é especializada na leitura de strings de arquivos de texto.

A classe `java.io.FilterReader`, abstrata, embora oferecendo funcionalidades adicionais, é concretamente implementada na classe `java.io.PushBackReader`, cujo uso é bastante restrito.

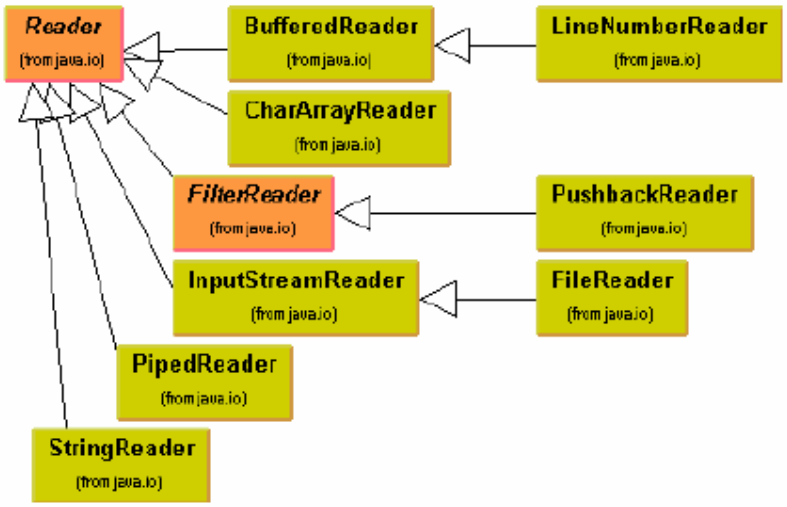

Figura 10. 2 - Hierarquia de Classes de java.io.Reader

Os construtores e métodos oferecidos pela classe `java.io.Reader` são:

Método	Descrição
Reader()	Constrói um objeto tipo **Reader** sobre o objeto **InputStream** especificado.
close()	Fecha a *stream* e libera os recursos associados do sistema.
mark(int)	Marca a posição atual da *stream*. O valor fornecido indica a validade da marca para leituras adicionais.
markSupported()	Verifica se as operações **mark** e **reset** são suportadas.
read()	Lê o próximo byte disponível.
read(byte[])	Preenche o vetor de bytes fornecido como argumento.
read(byte[], int, int)	Preenche o vetor de bytes fornecido como argumento entre as posições especificadas.
ready()	Verifica se a *stream* está pronta para leitura.
reset()	Reposiciona a *stream* na última posição determinada por **mark**.
skip(long)	Descarta a quantidade de bytes especificada.

Tabela 10.4 - Métodos da Classe java.io.Reader

10.3.4 – A Família java.io.OutputStream

Analogamente a classe `java.io.InputStream`, a `java.io.OutputStream` é uma classe abstrata, ou seja, não pode ser usada diretamente, mas origina várias diferentes classes especializadas que podem ser usadas em conjunto para prover a funcionalidade desejada. Os tipos de `OutputStream`, idênticos aos descritos na Tabela 10.1, permitem

o envio de dados para diferentes estruturas de dados, utilizando classes especializadas que oferecem tratamento para cada um destes tipos. Como `InputStream`, as classes derivadas de `OutputStream` são apropriadas para tratamento de arquivos orientados a byte e que utilizem o formato ASCII comum. Para cada classe da família `OutputStream` encontramos um correspondente na família `InputStream`. A hierarquia de classes da família `java.io.OutputStream` pode ser vista na Figura 10.3.

A classe `java.io.ByteArrayOutputStream` permite a criação de um buffer em memória, onde seu construtor recebe o tamanho inicial do buffer, podendo ser usado em conjunto com a classe `java.io.FilterOutputStream` para prover uma interface útil. Para envio de dados para arquivos utilizamos a classe `java.io.FileOutputStream`, cujo construtor recebe um objeto `String`, `File` ou `FileDescriptor` contendo informações sobre o arquivo a ser escrito. Outra vez a associação com um objeto da classe `java.io.FilterInputStream` permite o acesso a uma interface útil para programação.

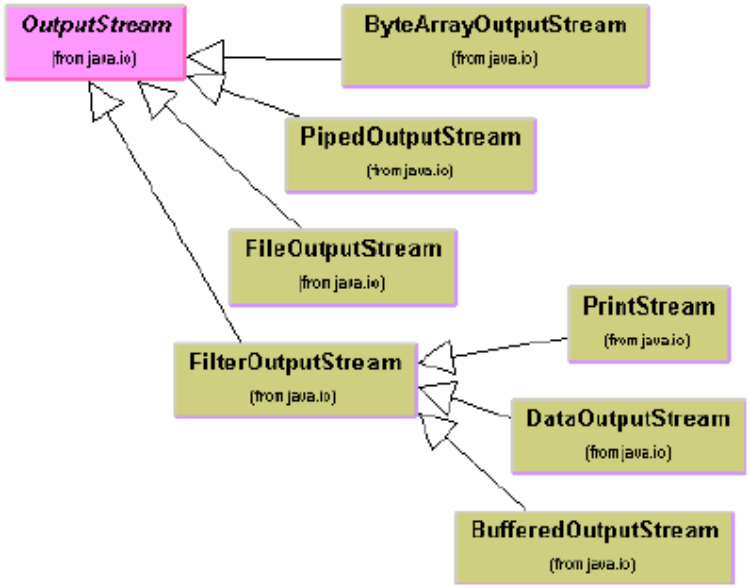

Figura 10.3 - Hierarquia de Classes de java.io.OutputStream

A classe `java.io.PipedOutputStream` é utilizada em conjunto com a classe `java.io.PipedInputStream` para a implementação de pipes (dutos de dados).

A classe `java.io.FilterOutputStream` serve como base para implementação de três outras classes especializadas, como mostra a Tabela 10.5.

Tipo	Descrição
DataOutputStream	Permite o armazenamento de dados formatados como tipos primitivos (char, int, float etc.)
PrintStream	Permite a produção de dados formatados como tipos primitivos orientado a exibição.
BufferedOutputStream	Permite otimizar as operações de escrita, realizadas em blocos.

Tabela 10.5 - Tipos de FilterOutputStream

A classe `java.io.FilterOutputStream` possui os seguintes construtores e métodos:

Método	Descrição
FilterOutputStream(OutputStream)	Cria um objeto **FilterOutputStream** a partir da **OutputStream** especificada.
close()	Fecha a *stream* e libera os recursos associados.
flush()	Força a escrita de qualquer dado ainda presente no *buffer* da *stream*.
write(byte[])	Escreve o *array* de bytes na *stream*.
write(byte[], int, int)	Escreve as posições especificadas do *array* de bytes na *stream*.
write(int)	Escreve o bytes especificado na *stream*.

Tabela 10.6 - Métodos da Classe java.io.FilterOutputStream

As classes derivadas de `java.io.FilterOutputStream` adicionam funcionalidade ou especializam ainda mais as operações contidas nesta classe, tal como a classe `java.io.PrintStream` que oferece os métodos `print` e `println` para obtenção de saída formatada de dados. A seguir um exemplo simples de uso destas classes na criação de um arquivo de saída:

```
//UsaOutputStream.java
import java.io.*;
public class UsaOutputStream {
  public static void main(String [] args) {
    try {
      DataOutputStream out =
        new DataOutputStream(
          new BufferedOutputStream(
            new FileOutputStream("Arquivo.ext")));
      out.writeBytes("Um frase simples");
      out.writeDouble(265.34554);
      out.close();
    } catch (IOException exc) {
      System.out.println("Erro de IO");
      exc.printStackTrace();
    }
  }
}
```

Exemplo 10. 2 - Uso de OutputStream

Note que são enviados tipos de dados diferentes (uma string e um valor real) para a stream sendo convertidos para o formato texto pelo objeto `DataOutputStream`.

10.3.5 – A Família java.io.Writer

A classe `java.io.Writer` é uma classe abstrata que serve como superclasse para a criação de outras, mais especializadas na escrita de caracteres a streams. Todas as suas subclasses devem implementar os métodos `close`, `flush` e `write`, que são suas operações básicas. Como para as outras classe abstratas do pacote `java.io`, as subclasses geralmente implementam tais métodos de forma mais especializada ou adicionam novas

funcionalidades. Outro aspecto importante é que a família de classe `java.io.Writer` oferece suporte para o UNICODE permitindo a internacionalização de aplicações.

Na Figura 10.4 temos a hierarquia de classes de `java.io.Writer`.

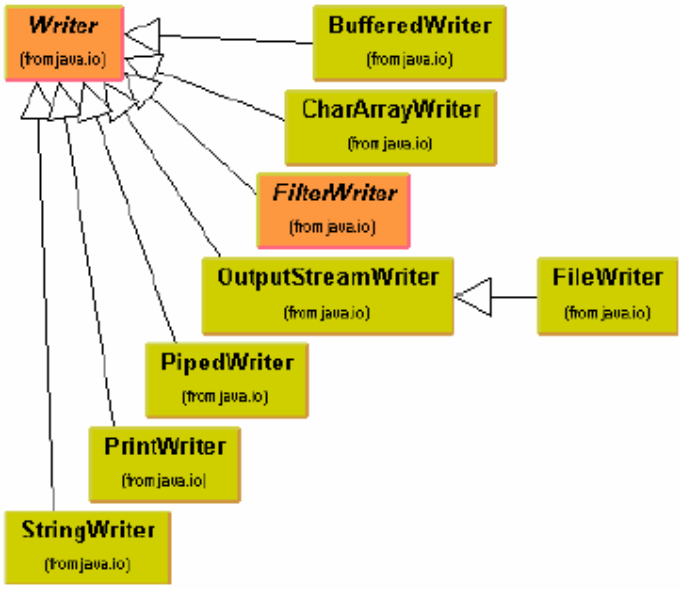

Figura 10. 4 - Hierarquia de Classes de java.io.Writer

Nela podemos observar a existência de diversas classes especializadas, entre elas: `java.io.BufferedWriter` para realização de operações de escrita através de uma stream bufferizada, `java.io.OutputStreamWriter` destinada a operações com arquivos, `java.io.PipedWriter` utilizada em conjunto com `java.io.PipedReader` para implementação de pipes e `java.io.PrintWriter` especializada em produção de saída formatada de dados.

Os principais métodos e construtores da classe `java.io.Writer` são:

Método	Descrição
Writer(Object)	Cria um objeto **Writer** sincronizado no objeto dado como argumento.
close()	Fecha a *stream*, gravando os dados existentes no *buffer* e liberando os recursos associados.
write(char[])	Escreve um *array* de caracteres na *stream*.
write(char[], int, int)	Escreve a porção especificada de um *array* de caracteres na *stream*.
write(int)	Escreve um caractere na *stream*.
write(String)	Escreve uma *string* na *stream*.
write(String, int, int)	Escreve a porção especificada de uma *string* de caracteres na *stream*.

Tabela 10.7 - Métodos da Classe java.io.Writer

Segue um exemplo de uso de streams com classes derivadsa de `Reader` e `Writer`. A classe ArquivoTexto é usada para permitir a leitura ou escrita de linhas em arquivos texto:

```
// ArquivoTexto.java
import java.io.*;

public class ArquivoTexto {
  File arquivo;
  BufferedReader in;
  PrintWriter out;

  ArquivoTexto(String nomeArq) {
     arquivo = new File (nomeArq);
  }
  ArquivoTexto(String nomeDir, String nomeArq) {
     arquivo = new File (nomeDir, nomeArq);
  }
  void abreLeitura() {
     try {
       in = new BufferedReader(new
                     FileReader(arquivo));
     } catch (FileNotFoundException e) {
       System.err.println ("Arquivo nao encontrado: "
                     + arquivo);
       e.printStackTrace();
     }
  }
  void abreEscrita() {
     try {
       out = new PrintWriter(new
                     FileWriter(arquivo, true));
     } catch (IOException e) {
       e.printStackTrace();
     }
  }
  void fechaLeitura() {
     try {
       in.close();
     } catch (IOException e) {
       e.printStackTrace();
     }
  }
  void fechaEscrita() {
       out.close();
  }
  String leProxLinha() {
     String s = null;
     try {
       s = in.readLine();
     } catch (IOException e) {
        e.printStackTrace();
        s=null;
     }
     return s;
```

```
  }
  void gravaProxLinha(String linha) {
     out.println(linha);
  }
}
```

Exemplo 10. 3 - A Classe ArquivoTexto

Segue um exemplo de aplicação simples capaz de ler o conteúdo de um arquivo texto e criar um novo arquivo texto composto pelas mesmas linhas, mas prefixadas e pós-fixadas por delimitadores:

```
// UsaArquivosTexto.java
public class UsaArquivosTexto {
  public static void main (String [] args) {
    System.out.print (
       "Entre com nome do arquivo de entrada: ");
    String nomeEntrada = Console.readString();
    System.out.print (
       "Entre com nome do arquivo de saida: ");
    String nomeSaida = Console.readString();
    ArquivoTexto entrada = new
       ArquivoTexto (nomeEntrada);
    ArquivoTexto saida = new ArquivoTexto (nomeSaida);
    entrada.abreLeitura();
    saida.abreEscrita();
    String s;
    while (null != (s = entrada.leProxLinha())) {
      saida.gravaProxLinha(" === " + s + " === ");
    }
    entrada.fechaLeitura();
    saida.fechaEscrita();
  }
}
```

Exemplo 10. 4- Usando a Classe ArquivoTexto

10.4 – Acesso Direto a Arquivos

As classes para tratamento de streams vistas anteriormente permitiam apenas o acesso sequencial aos dados, isto é, a leitura de todos os dados do início do arquivo até o dado desejado. Em algumas situações o acesso sequencial aos dados existentes em arquivos se mostra inadequado pois pode ser necessária a leitura de muitos dados antes da informação requerida ser acessada, tornando tal acesso lento e consumindo recursos de forma desnecessária.. Com acesso aleatório podemos acessar diretamente uma determinada posição de um arquivo, possibilitando operações de leitura e escrita mais eficientes e adequadas para tratamento de grande volume de dados.

A classe `java.io.RandomAccess` oferece suporte tanto para leitura como para escrita em posições aleatórias de um arquivo de dados. Em particular, é através dela que se consegue abrir um arquivo para leitura e gravação simultâneas (por exemplo, para adicionar dados ao final de um arquivo existente). Esta classe também oferece meios da aplicação limitar as operações que podem ser realizadas no arquivo, garantindo sua integridade.

Essencialmente, esta classe funciona como uma `DataInputStream` miscigenada com uma `DataOutputStream` com métodos adicionais para identificar onde o cursor se encontra no arquivo, para mover o cursor e para determinar o tamanho do arquivo. Além disso, os construtores desta classe requerem um argumento indicando se o arquivo será aberto para leitura (`"r"`) ou para leitura e escrita (`"rw"`). Não existe, portanto, suporte para abrir arquivos somente para escrita.

Os principais construtores e métodos disponíveis nesta classe são:

Método	Descrição
RandomAcessFile(File, String)	Constrói um objeto desta classe recebendo um objeto **File** especificando o arquivo e uma *string* determinando o modo de acesso.
RandomAcessFile(String, String)	Constrói um objeto desta classe recebendo um objeto **String** especificando o arquivo e uma *string* determinando o modo de acesso.
close()	Fecha a *stream*, gravando dados pendentes e liberando os recursos associados.
getFilePointer()	Obtêm a posição corrente dentro da *stream*.
lenght()	Obtêm o tamanho total do arquivo.
read(byte[])	Lê o *array* de bytes fornecido.
readBoolean(), readByte(), readChar(), readFloat(), readInt(), readLong()	Lê um valor no formato especificado da *stream*.
readLine()	Lê uma linha do arquivo associado a *stream*.
seek(long)	Movimenta o cursor para a posição especificada.
skipBytes(long)	Avança o cursor da quantidade de bytes indicada.
write(byte[])	Escreve o *array* de bytes fornecido na *stream*.
writeBoolean(boolean), writeByte(byte), writeChar(chat), writeFloat(float), writeInt(int), writeLong(long)	Escreve o valor do tipo especificado na *stream*..

Tabela 10.8 - Métodos da Classe java.io.RandomAccessFile

Como pode ser observado, esta classe oferece inúmeros métodos para a escrita e leitura de tipos primitivos facilitando assim o seu emprego. No entanto, para utilização de arquivos com acesso aleatório com dados de tipo diferente dos primitivos devemos trabalhar com uma classe que encapsule os dados escrevendo e lendo os mesmos através dos métodos `read` e `write`.

No Exemplo 96 temos uma classe que demonstra a criação e leitura de dados através da classe java.io.RandomAccessFile. Um arquivo de dados de teste pode ser criado através da aplicação. Registros individualmente especificados pelo usuário podem ser lidos e apresentados na área central da aplicação.

```
// UsaRandom.java
import java.io.*;
public class UsaRandom {
  public static void main(String args[]) {
    System.out.print ("Entre com nome do arquivo: ");
    String nome = Console.readString();
```

```
    try {
      RandomAccessFile f =
           new RandomAccessFile (nome, "rw");
      try {
        f.seek(f.length());
        f.writeByte('\n');
        for (int i = 0; i < 25; i++) {
          f.writeByte('+');
        }
      } catch (IOException e) {
        System.out.println("Falhou!!!");
      } finally {
        f.close();
      }
    } catch (IOException e) {
      System.out.println("Nao abriu!!!");
    }
  }
}
```

Exemplo 10. 5 - Acesso Direto a Arquivos

10.5 – Serialização

Até agora vimos como enviar e recuperar valores de tipos primitivos, vetores e strings em streams, mas não como fazer isso para os objetos de outras classes. Uma maneira de fazer isso é incluindo métodos que implementem estas operações na classe do objeto através do encapsulamento das operações de envio e recuperação de tipos primitivos. Esses métodos devem levar em conta que:

1. O objeto deve ser convertido de sua representação encadeada na memória primária para uma representação seqüencial na memória secundária. Como os valores (absolutos) dos ponteiros na memória não terão significado na próxima vez que o programa for executado, este ponteiros devem ser relativizados quando armazenados.
2. Ao restaurar um objeto da memória secundária, os ponteiros devem ser ajustados de modo a respeitar as relações existentes anteriormente entre os atributos do objeto.

A maior dificuldade com essa abordagem é que ela requer uma boa dose de trabalho do programador. Objetos podem ser compostos por emaranhados de outros objetos e o programador terá de escrever código capaz de percorrer e restabelecer estes emaranhados.

A partir da versão 1.1, Java adicionou uma interessante característica, chamada serialização, que permite a qualquer objeto que implemente a interface `Serializable` ser transformado em uma seqüência de bytes e ser posteriormente recuperado tal como o objeto original. Isto é verdadeiro até mesmo numa rede, o que implica que o mecanismo de serialização automaticamente compensa as diferenças entre sistemas operacionais. Em outras palavras, um objeto criado em um ambiente Windows e serializado pode ser enviado pela rede para uma máquina Unix onde será reconstruído corretamente. Portanto, o programador não necessita se preocupar sobre as diferentes representações nas diferentes plataformas, sobre a relativização dos dados e quaisquer outros detalhes.

Serializar um objeto é muito fácil, desde que a classe do objeto implemente a interface `Serializable`. Esta interface é um tipo de interface especial de Java, pois não possui

métodos. Ela funciona, portanto, como um sinalizador apenas. Assim, para declarar uma classe que implemente a interface `Serializable` basta indicar isso no cabeçalho de declaração da classe. Por exemplo:

```
class Data implements Serializable {
  // código da classe
  // …
}
```

Uma vez que a classe implemente esta interface, para serializar seus objetos basta criar um tipo de OutputStream e abrigá-lo em um objeto ObjectOutputStream. Neste ponto, é necessário chamar o método writeObject() para que o objeto seja serializado e enviado para a OutputStream. O processo reverso, de recuperação, requer que um tipo de InputStream seja abrigado em um ObjectInputStream e chamar readObject().

A característica especialmente interessante da serialização de objetos é que ela automaticamente segue as referências contidas no objeto e salva todos os objetos referenciados. Desta maneira, o programador fica totalmente liberado de ter de programar estas tarefas. O exemplo seguinte mostra como a serialização de objetos pode ser implementada:

```
//Persiste.java
import java.io.*;
class Dado implements Serializable {
   private int i;
   Dado(int x) { i = x; }
   public String toString() {
     return Integer.toString(i);
   }
}

public class Persiste implements Serializable {
   private static int r() {
      return (int)(Math.random() * 10);
   }
   private Dado[] d = {
      new Dado(r()), new Dado(r()), new Dado(r())
   };
   private  Persiste prox;
   private char c;
   Persiste(int i, char x) {
     System.out.println("Construtor Persiste: " + i);
     c = x;
     if(--i > 0)
       prox = new Persiste(i, (char)(x + 1));
   }
   Persiste() {
     System.out.println("Construtor Default ");
   }
   public String toString() {
     String s = ":" + c + "(";
     for(int i = 0; i < d.length; i++)
       s += d[i].toString();
```

```
      s += ")";
      if(prox != null)
        s += prox.toString();
      return s;
  }
  public static void main(String[] args) {
    Persiste w = new Persiste(6, 'a');
    System.out.println("w = " + w);
    try {
      ObjectOutputStream out =
         new ObjectOutputStream(
            new FileOutputStream("persiste.out"));
      out.writeObject("Armazena Persiste");
      out.writeObject(w);
      out.close();
      ObjectInputStream in =
         new ObjectInputStream(
            new FileInputStream("persiste.out"));
      String s = (String)in.readObject();
      Persiste w2 = (Persiste)in.readObject();
      System.out.println(s + ", w2 = " + w2);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Exemplo 10. 6 - Serialização