Helder da Rocha
www.argonavis.com.br

Assuntos abordados

- *Este módulo explora os componentes mais importantes do pacote java.io e outros recursos da linguagem relacionados à E/S e arquivos*
- *A classe File, que representa arquivos e diretórios*
- *Objetos que implementam entrada e saída*
  - *InputStream e OutputStream, Readers e Writers*
  - *Compressão com GZIP streams*
  - *FileChannels*
- *Objeto que implementa arquivo de acesso aleatório*
  - *RandomAccessFile*
- *Recursos de serialização básica*
  - *Serializable, ObjectOutputStream e ObjectInputStream*
- *Logging: fundamentos*

O pacote java.io

- Oferece abstrações que permitem ao programador lidar com arquivos, diretórios e seus dados de uma maneira independente de plataforma
  - File, RandomAccessFile
- Oferecem recursos para facilitar a manipulação de dados durante o processo de leitura ou gravação
  - bytes sem tratamento
  - caracteres Unicode
  - dados filtrados de acordo com certo critério
  - dados otimizados em buffers
  - leitura/gravação automática de objetos
- Pacote java.nio (New I/O): a partir do J2SDK 1.4.0
  - Suporta mapeamento de memória e bloqueio de acesso

A classe File

- Usada para representar o sistema de arquivos
  - É apenas uma abstração: a existência de um objeto File não significa a existência de um arquivo ou diretório
  - Contém métodos para testar a existência de arquivos, para definir permissões (nos S.O.s onde for aplicável), para apagar arquivos, criar diretórios, listar o conteúdo de diretórios, etc.
- Alguns métodos
  - String getAbsolutePath()
  - String getParent(): retorna o diretório (objeto File) pai
  - boolean exists()
  - boolean isFile()
  - boolean isDirectory()
  - boolean delete(): tenta apagar o diretório ou arquivo
  - long length(): retorna o tamanho do arquivo em bytes
  - boolean mkdir(): cria um diretório com o nome do arquivo
  - String[] list(): retorna lista de arquivos contido no diretório

File: exemplo de uso

```
File diretorio = new File("c:\tmp\cesto");
diretorio.mkdir(); // cria, se possível
File arquivo = new File(diretorio, "lixo.txt");
FileOutputStream out =
            new FileOutputStream(arquivo);
// se arquivo não existe, tenta criar
out.write( new byte[]{'l','i','x','o'} );

File subdir = new File(diretorio, "subdir");
subdir.mkdir();
String[] arquivos = diretorio.list();
for (int i = 0; arquivos.length; i++) {
   File filho = new File(diretorio, arquivos[i]);
   System.out.println(filho.getAbsolutePath());
}
if (arquivo.exists()) {
   arquivo.delete();
}
```

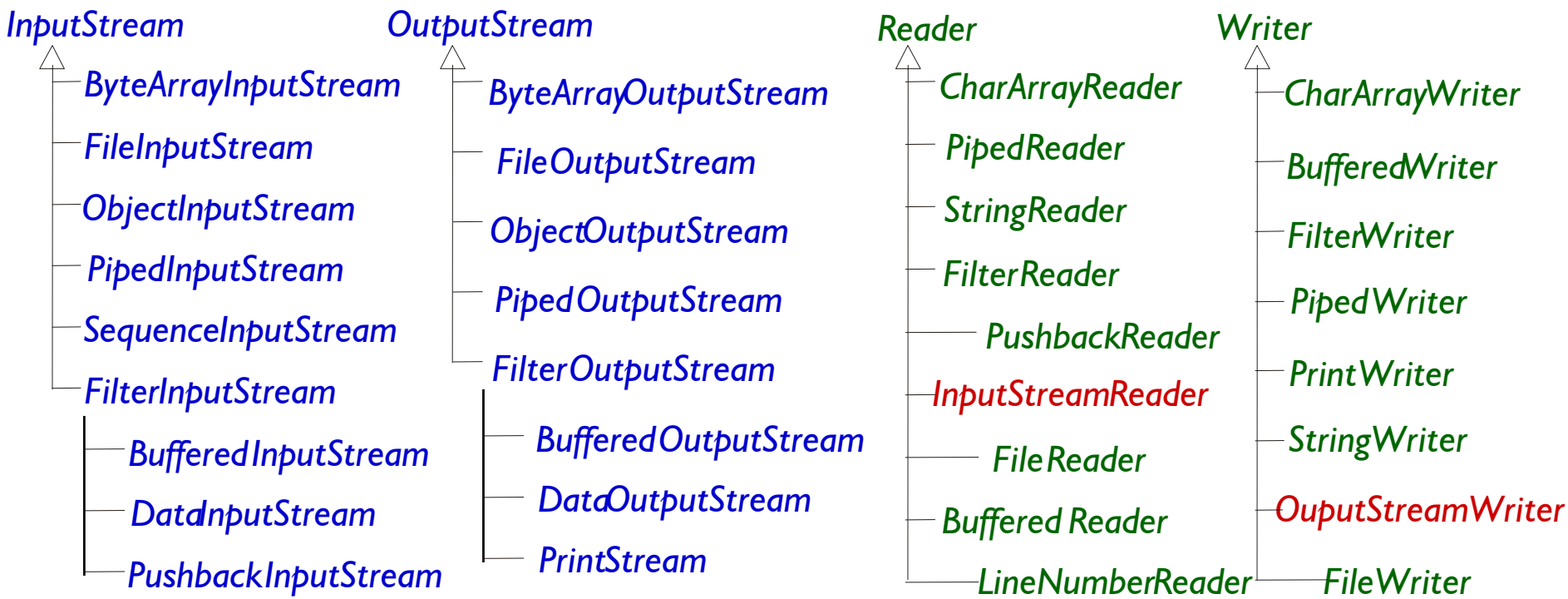
O bloco de código acima precisa tratar IOException

Fluxos de Entrada e Saída

- Há várias fontes de onde se deseja ler ou destinos para onde se deseja gravar ou enviar dados
  - Arquivos
  - Conexões de rede
  - Console (teclado / tela)
  - Memória
- Há várias formas diferentes de ler/escrever dados
  - Seqüencialmente / aleatoriamente
  - Como bytes / como caracteres
  - Linha por linha / palavra por palavra, ...
- APIs Java para I/O oferecem objetos que abstraem fontes/destinos (nós) e fluxos de bytes e caracteres

Classes e interfaces para fluxos de E/S

- Dois grupos:
  - e/s de bytes: InputStream e OutputStream
  - e/s de chars: Reader e Writer



E/S de bytes

- *InputStream*
  - *Classe genérica (abstrata) para lidar com fluxos de bytes de entrada e nós de fonte (dados para leitura).*
  - *Método principal: read()*
- *OutputStream*
  - *Classe genérica (abstrata) para lidar com fluxos de bytes de saída e nós de destino (dados para gravação).*
  - *Método principal: write()*
- *Principais implementações*
  - *Nós (fontes): FileInputStream (arquivo), ByteArrayInputStream (memória) e PipedInputStream (pipe).*
  - *Processamento de entrada: FilterInputStream (abstract) e subclasses*
  - *Nós (destinos): FileOutputStream (arquivo), ByteArrayOutputStream (memória) e PipedOutputStream (pipe).*
  - *Processamento de saída: FilterOutputStream (abstract) e subclasses.*

Métodos de InputStream e OutputStream

- Principais métodos de InputStream
  - int read(): retorna um byte (ineficiente)
  - int read(byte[] buffer): coloca bytes lidos no vetor passado como parâmetro e retorna quantidade lida
  - int read(byte[] buffer, int offset, int length): idem
  - void close(): fecha o stream
  - int available(): número de bytes que há para ler agora
- Métodos de OutputStream
  - void write(int c): grava um byte (ineficiente)
  - void write(byte[] buffer)
  - void write(byte[] buffer, int offset, int length)
  - void close(): fecha o stream (essencial)
  - void flush(): esvazia o buffer

Exemplo de leitura e gravação de arquivo

- *Trecho de programa que copia um arquivo**

```
String nomeFonte = args[0];
String nomeDestino = args[1];
File fonte = new File(nomeFonte);
File destino = new File(nomeDestino);
if (fonte.exists() && !destino.exists()) {
   FileInputStream in = new FileInputStream(fonte);
   FileOutputStream out = new FileOutputStream(destino);
   byte[] buffer = new byte[8192];
   int length = in.read(buffer);
   while ( length != -1) {
      out.write(buffer, 0, length);
      in.read(buffer);
   }
   in.close();
   out.flush();
   out.close();
}
```

-1 sinaliza EOF

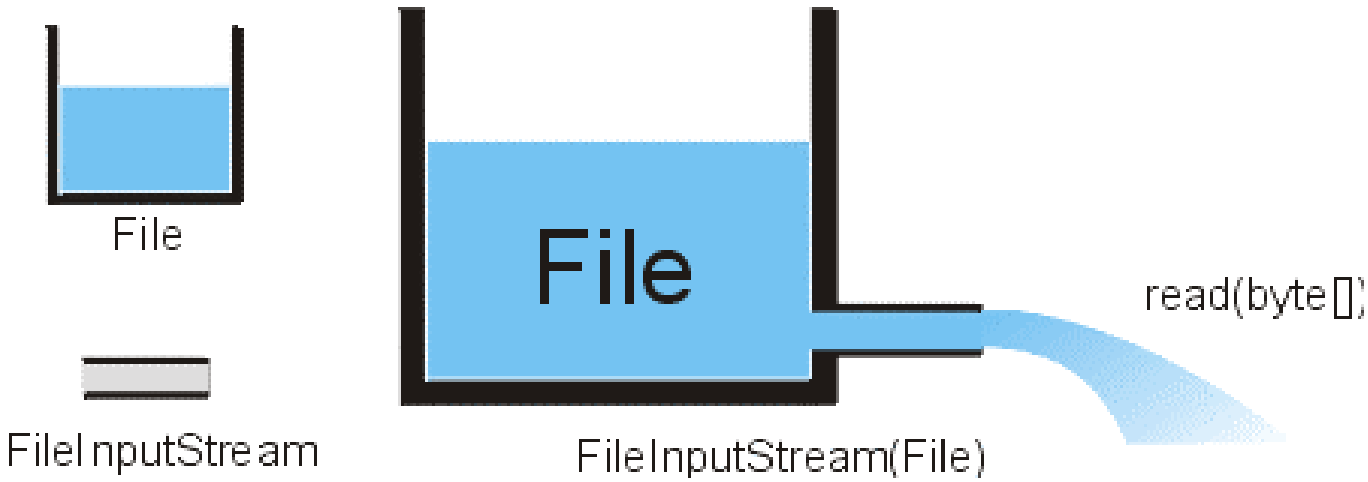
Grava apenas os bytes lidos (e não o buffer inteiro)

* *Método e blocos try-catch (obrigatórios) foram omitidos para maior clareza.*

FilterStreams

- Implementam o padrão de projeto Decorator
  - São concatenados em streams primitivos oferecendo métodos mais úteis com dados filtrados
- FilterInputStream: recebe fonte de bytes e oferece métodos para ler dados filtrados. Implementações:
  - DataInputStream: readInt(), readUTF(), readDouble()
  - BufferredInputStream: read() mais eficiente
  - ObjectOutputStream: readObject() lê objetos serializados
- FilterOutputStream: recebe destino de bytes e escreve dados via filtro. Implementações:
  - DataOutputStream: writeUTF(), writeInt(), etc.
  - BufferedOutputStream: write() mais eficiente
  - ObjectOutputStream: writeObject() serializa objetos
  - PrintStream: classe que implementa println()

Exemplo: leitura de um stream fonte (arquivo)



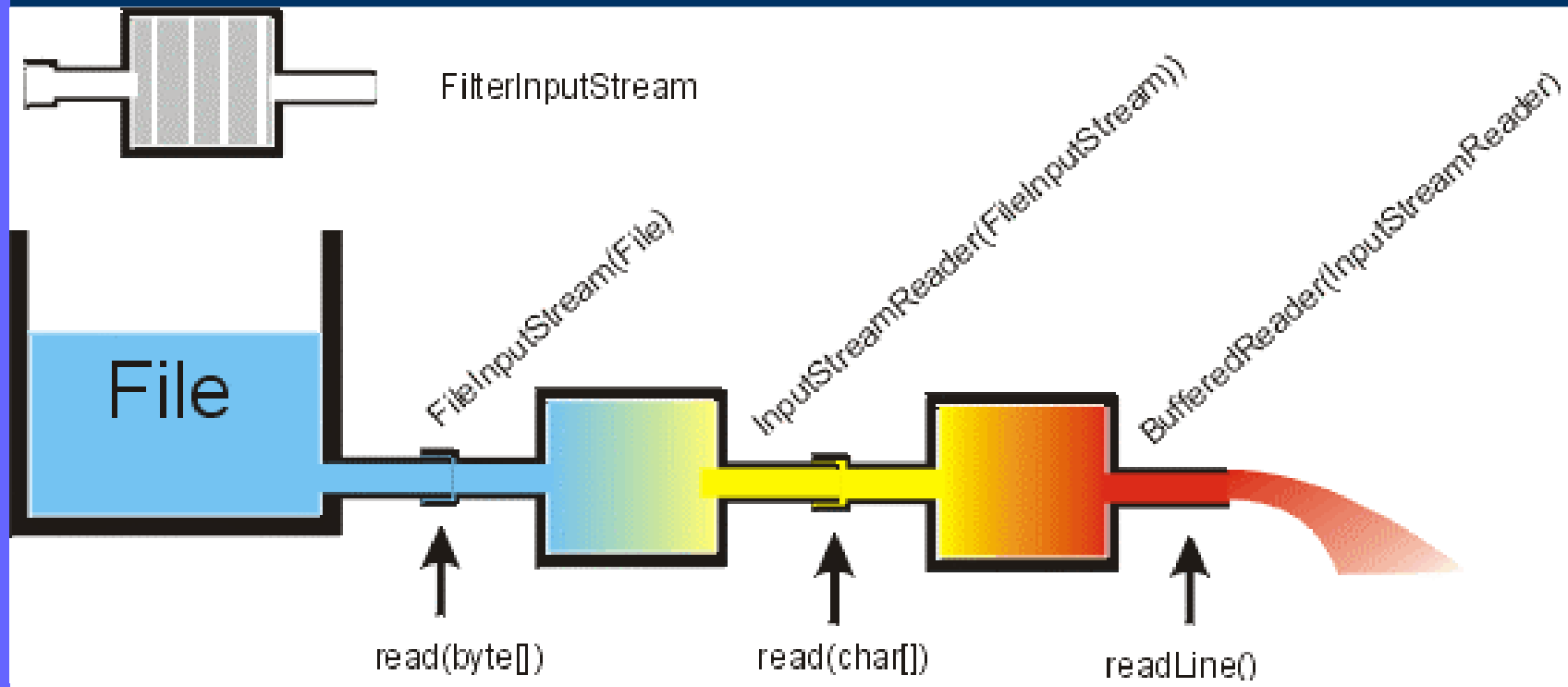

```
// objeto do tipo File
File tanque = new File("agua.txt");

// referência FileInputStream
// cano conectado no tanque
FileInputStream cano =
                  new FileInputStream(tanque);

// lê um byte a partir do cano
byte octeto = cano.read();
```

Usando filtro para ler char

- *InputStreamReader é um filtro que converte bytes em chars*
  - *Para ler chars de um arquivo pode-se usar diretamente um FileWriter em vez de concatenar os filtros abaixo.*

```
// objeto do tipo File
File tanque = new File("agua.txt");

// referência FileInputStream
// cano conectado no tanque
FileInputStream cano =
                  new FileInputStream(tanque);

// filtro chf conectado no cano
InputStreamReader chf =
              new InputStreamReader(cano);

// lê um char a partir do filtro chf
char letra = chf.read();
```

Usando outro filtro para ler linha




```
// filtro chf conectado no cano
InputStreamReader chf = new InputStreamReader(cano);
// filtro br conectado no chf
BufferedReader br = new BufferedReader (chf);
// lê linha de texto a de br
String linha = br.readLine();
```

E/S de caracteres

- Reader
  - Classe abstrata para lidar com fluxos de caracteres de entrada: método read() lê um caractere (16 bits) por vez
- Writer
  - Classe abstrata para lidar com fluxos de bytes de saída: método write() grava um caractere (16 bits) por vez
- Principais implementações
  - Nós (destinos): FileWriter (arquivo), CharArrayWriter (memória), PipedWriter (pipe) e StringWriter (memória).
  - Processamento de saída: FilterWriter (abstract), BufferedWriter, OutputStreamWriter (conversor de bytes para chars), PrintWriter
  - Nós (fontes): FileReader (arquivo), CharArrayReader (memória), PipedReader (pipe) e StringReader (memória).
  - Processamento de entrada: FilterReader (abstract), BufferedReader, InputStreamReader (conversor bytes p/ chars), LineNumberReader

Métodos de Reader e Writer

- Principais métodos de Reader
  - int read(): lê um char (ineficiente)
  - int read(char[] buffer): coloca chars lidos no vetor passado como parâmetro e retorna quantidade lida
  - int read(char[] buffer, int offset, int length): idem
  - void close(): fecha o stream
  - int available(): número de chars que há para ler agora
- Métodos de Writer
  - void write(int c): grava um char (ineficiente)
  - void write(char[] buffer)
  - void write(char[] buffer, int offset, int length)
  - void close(): fecha o stream (essencial)
  - void flush(): esvazia o buffer

Leitura e gravação de texto com buffer

- *A maneira mais eficiente de ler um arquivo de texto é usar FileReader decorado por um BufferedReader. Para gravar, use um PrintWriter decorando o FileWriter*

```
BufferedReader in = new BufferedReader(
    new FileReader("arquivo.txt"));
StringBuffer sb =
    new StringBuffer(new File("arquivo.txt").length());
String linha = in.readLine();
while( linha != null ) {
    sb.append(linha).append('\n');
    linha = in.readLine();
}
in.close();
String textoLido = sb.toString();
// (...)
PrintWriter out = new PrintWriter(
    new FileWriter("ARQUIVO.TXT"));
out.print(textoLido.toUpperCase());
out.close();
```

Leitura da entrada padrão e memória

- A entrada padrão (System.in) é representada por um objeto do tipo InputStream.
- O exemplo abaixo lê uma linha de texto digitado na entrada padrão e grava em uma String. Em seguida lê seqüencialmente a String e imprime uma palavra por linha

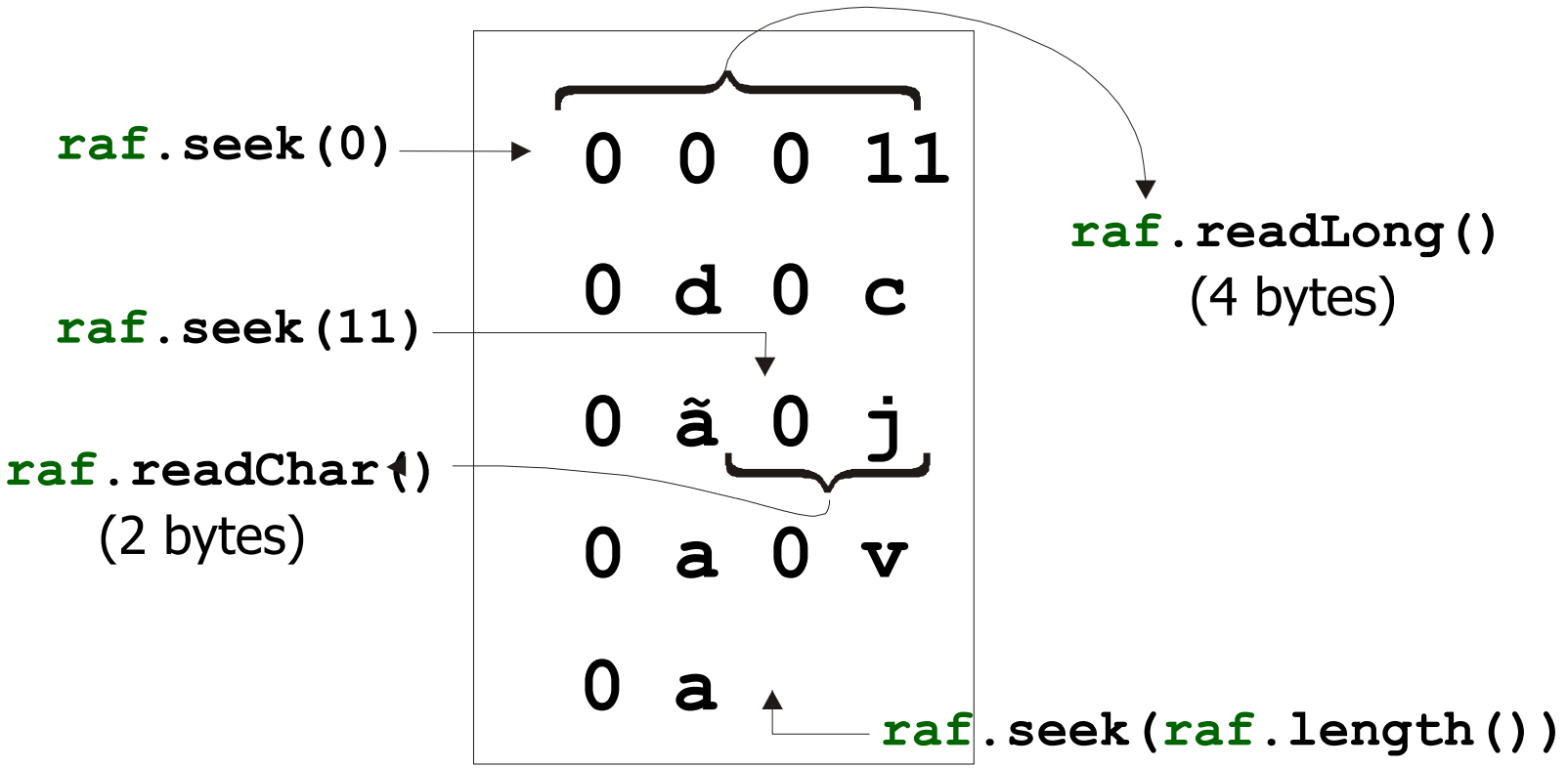
```
BufferedReader stdin = new BufferedReader(
    new InputStreamReader(System.in));
System.out.print("Digite uma linha:");
String linha = stdin.readLine());

StringReader rawIn = new StringReader(linha);
int c;
while((c = rawIn.read()) != -1)
  if ( c == ' ') System.out.println();
  else System.out.print((char)c);
}
```

RandomAccessFile

- Classe "alienígena": não faz parte da hierarquia de fluxos de dados do java.io
  - Implementa interfaces DataOutput e DataInput
  - Mistura de File com streams: não deve ser usado com outras classes (streams) do java.io
- Oferece acesso aleatório a um arquivo através de um ponteiro
- Métodos (DataOutput e DataInput) tratam da leitura e escrita de Strings e tipos primitivos
  - void seek(long)
  - readInt(), readBytes(), readUTF(), ...
  - writeInt(), writeBytes(), writeUTF(), ...

RandomAccessFile

```
RandomAccessFile raf =
        new RandomAccessFile ("arquivo.dat", "rw");
```



Exceptions

- *A maior parte das operações de E/S provoca exceções que correspondem ou são subclasses de IOException*
  - *EOFException*
  - *FileNotFoundException*
  - *StreamCorruptedException*
- *Para executar operações de E/S é preciso, portanto, ou capturar IOException ou repassar a exceção através de declarações throws nos métodos causadores*
- *Nos exemplos mostrados o tratamento de exceções foi omitido. Tipicamente, as instruções close() ocorrem em um bloco try-catch dentro de um bloco finally*

```
try { ... } finally {
    try { stream.close(); } catch (IOException e) {}
}
```

Não adianta saber se o fechamendo to stream falhou

Serialização

- Java permite a gravação direta de objetos em disco ou seu envio através da rede
  - Para isto, o objeto deve declarar implementar java.io Serializable
- Um objeto Serializable poderá, então
  - Ser gravado em qualquer stream usando o método writeObject() de ObjectOutputStream
  - Ser recuperado de qualquer stream usando o método readObject() de ObjectInputStream
- Um objeto serializado é um grafo que inclui dados da classe e todas as suas dependências
  - Se a classe ou suas dependências mudar, o formato usado na serialização mudará e os novos objetos serão incompatíveis com os antigos (não será mais possível recuperar arquivos gravados com a versão antiga)

Exemplo: gravação e leitura de objetos

```
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream(armario)
);

Arco a = new Arco();
Flecha f = new Flecha();

// grava objeto Arco em armario
out.writeObject(a);

// grava objeto flecha em armario
out.writeObject(f);
```

Gravação de objetos

Leitura de objetos na mesma ordem

```
ObjectInputStream in = new ObjectInputStream(
    new FileInputStream(armario)
);

// recupera os dois objetos
// método retorna Object (requer cast)
Arco primeiro = (Arco)in.readObject();
Flecha segundo = (Flecha)in.readObject();
```

ZIP e JAR

- Os pacotes *java.util.zip* e *java.util.jar* permitem comprimir dados e colecionar arquivos mantendo intactas as estruturas de diretórios
- Vantagens
  - Menor tamanho: maior eficiência de E/S e menor espaço em disco
  - Menos arquivos para transferir pela rede (também maior eficiência de E/S)
- Use classes de ZIP e JAR para coleções de arquivos
  - ZipEntry, ZipFile, ZipInputStream, etc.
- Use streams GZIP para arquivos individuais e para reduzir tamanho de dados enviados pela rede

Exemplo com GZIP streams

- *GZIP usa o mesmo algoritmo usado em ZIP e JAR mas não agrupa coleções de arquivos*
  - *GZIPOutputStream comprime dados na gravação*
  - *GZIPInputStream expande dados durante a leitura*
- *Para usá-los, basta incluí-los na cadeia de streams:*

```
ObjectOutputStream out = new ObjectOutputStream(
                new java.util.zip.GZIPOutputStream(
                   new FileOutputStream(armario) ) );

Objeto gravado = new Objeto();
out.writeObject(gravado);

// (...)

ObjectInputStream in = new ObjectInputStream(
                new java.util.zip.GZIPInputStream(
                   new FileInputStream(armario) ) );

Objeto recuperado = (Objeto)in.readObject();
```

FileChannel

- *Novidade no J2SDK 1.4*
- *Permite ler e gravar arquivos, mapeando memória e bloqueando acesso (afeta performance)*
  - *Mapeamento permite abrir o arquivo como se fosse um vetor, usando a classe java.nio.ByteBuffer. Ideal para ler arquivos consistindo de registros de tamanho fixo.*
  - *É preciso importar java.nio.* e java.nio.channels.*;*
- *Exemplo: ler registro de arquivo de registros fixos*

```
FileInputStream stream = new FileInputStream("a.txt");
FileChannel in = stream.getChannel();
int len = (int) in.size();
ByteBuffer map = in.map(FileChannel.MapMode.READ_ONLY, 0, len);
final int TAM = 80; // tamanho de cada registro: 80 bytes
byte[] registro = new byte[TAM]; //array p/ guardar 1 registro
map.position( 5 * TAM ); // posiciona antes do 5o. registro
map.get( registro ); // preenche array com dados encontrados
```

Logging

- Recurso introduzido no J2SDK 1.4
- Oferece um serviço que gera relatórios durante a execução de uma aplicação. Os relatórios são formados por eventos escolhidos pelo programador e podem ter como destino:
  - A tela (console), um arquivo, uma conexão de rede, etc.
- Os dados também podem ser formatados de diversas formas
  - Texto, XML e filtros
- As mensagens são classificadas de acordo com a severidade, em sete níveis diferentes. O usuário da aplicação pode configurá-la para exibir mais ou menos mensagens de acordo com o nível desejado
- Principais classes
  - java.util.logging.Logger e java.util.logging.Level

Logger

- *Para criar um Logger, é preciso usar seu construtor estático:*

```
Logger logger = Logger.getLogger("com.meupacote");
```

- *Os principais métodos de Logger encapsulam os diferentes níveis de detalhamento (severidade) ou tipos de informação. Métodos log() são genéricos e aceitam qualquer nível*
  - `config(String msg)`
  - `entering(String class, String method)`
  - `exiting(String class, String method)`
  - `fine(String msg)`
  - `finer(String msg)`
  - `finest(String msg)`
  - `info(String msg)`
  - `log(Level level, String msg)`
  - `severe(String msg)`
  - `throwing(String class, String method, Throwable e)`
  - `warning(String msg)`

Exemplo de Logging

- *Exemplo da documentação da Sun [J2SDK14]*

```
package com.wombat;
public class Nose{
    // Obtain a suitable logger.
    private static Logger logger =
        Logger.getLogger("com.wombat.nose");

    public static void main(String argv[]){
      // Log a FINE tracing message
      logger.fine("doing stuff");
      try{
         Wombat.sneeze();
      } catch (Error ex){ // Log the error
        logger.log(Level.WARNING,"trouble sneezing",ex);
      }
      logger.fine("done");
    }
}
```

Níveis de severidade

- As seguintes constantes da classe Level devem ser usadas para indicar o nível das mensagens gravadas
  - Level.OFF (não imprime nada no log)
  - Level.SEVERE (maior valor - imprime mensagens graves)
  - Level.WARNING
  - Level.INFO
  - Level.CONFIG
  - Level.FINE
  - Level.FINER
  - Level.FINEST (menor valor - imprime detalhamento)
  - Level.ALL (imprime tudo no log)
- Quanto maior o valor escolhido pelo usuário, menos mensagens são gravadas.

Exercícios

- *1. Escreva um programa que leia um arquivo de texto para dentro de uma janela de aplicação gráfica (TextArea)*
- *2. Faça um programa que leia um arquivo XML ou HTML e arranque todos os tags. Imprima na saída padrão.*
- *3. Aplicação da Biblioteca: Crie uma classe RepositorioDadosArquivo que implemente RepositorioDados mantenha arquivos armazenados em dois diretorios:*
  - *agentes/*
  - *publicacoes/*

  Cada diretório deverá armazenar um arquivo por registro. O nome do arquivo deve ser o codigo do registro e os dados devem estar guardados um em cada linha.
  - *Pode-se usar BufferedReader.readLine() para lê-los.*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br