Estrutura de dados - Arquivos

Fonte: Material cedido por Jacques Saïvé e tutorial da Sun (I/O)

Programação 2 – aula 23 e 24

Objetivos da seção

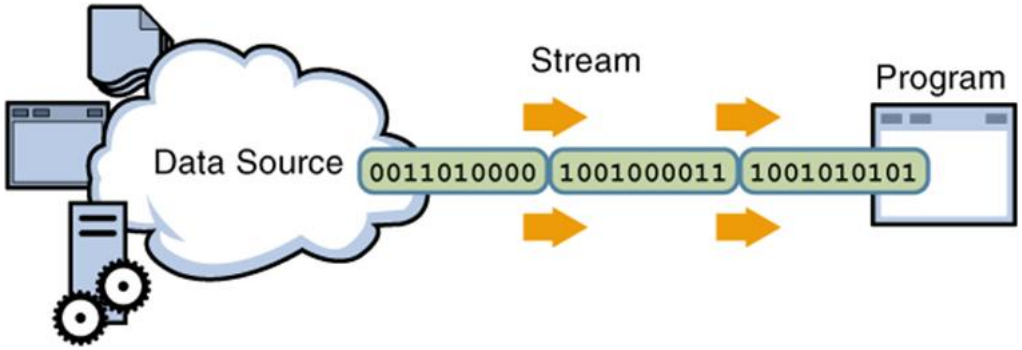
- Aprender a acessar depósitos de dados persistentes chamados "arquivos"
- Aprender a manipular arquivos de várias formas, incluindo para a serialização de objetos

Noções de stream (fluxo)

- Usaremos um conceito muito genérico para acessar arquivos: o Stream (fluxo)
- Um stream é uma sequência ordenada de dados com uma fonte e um destino

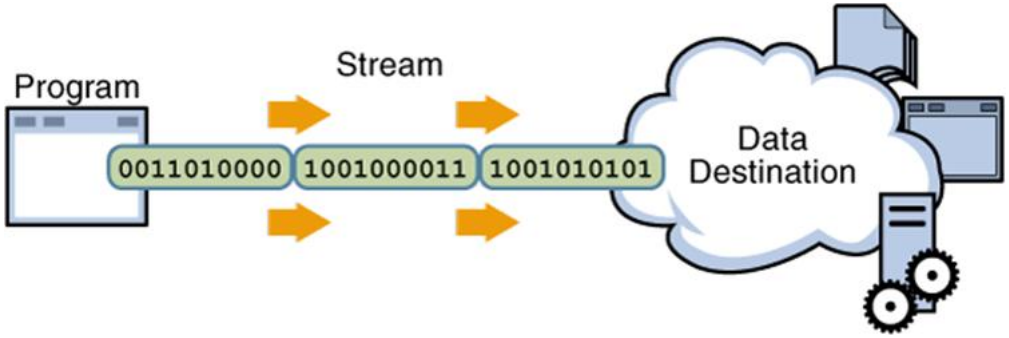
- O que é importante sobre o stream é que o acesso aos dados é sequencial
- Não é como acessar objetos na memória
  - Neste caso, acessamos qualquer objeto diretamente na memória usando a referência
- Se houver 1000 pedaços de informação num stream, acessamos cada pedaço, mas seqüencialmente, um de cada vez: "Me dê o próximo... me dê o próximo..."
- A noção de stream é importante, pois permite acessar vários tipos de "depósitos de informação"
  - Arquivos
  - Conexões de rede
  - Dispositivos (teclado, tela, impressora, fita de backup, câmera de vídeo, ...)
- Um stream pode ser caracterizado em várias dimensões

- **Direção**
  - Streams de entrada



i) Em Java, possuem a palavra "Input" na classe (ex. InputStream)
ii)O destino da informação é seu programa
iii) Isto é, seu programa lê informação do stream

  - Streams de saída



i) Em Java, possuem a palavra "Output" na classe (ex. OutputStream)
ii)A fonte da informação é seu programa
iii) Isto é, seu programa grava informação no stream

  - Streams bidirecionais

i) Em Java, possuem a palavra "RandomAccess" na classe (ex. RandomAccessFile)
ii)Seu programa lê e grava informação do/no stream

- **Tipo de informação lida**
  - Streams de bytes (caracterizados pela palavra Stream)

i) Leitura/gravação de bytes crus (nenhuma palavra adicional)
ii)Leitura/gravação de tipos nativos (palavra Data - DataInputStream)
iii) Leitura/gravação de objetos inteiros (palavra Object - ObjectOutputStream)
› Streams de caracteres Unicode (caracterizados pela palavra Reader ou Writer)

- **Fontes e destinos** (de onde vem/para onde vai a informação)
  - › Arquivo (palavra File)
    i) Para ter persistência de informação depois que seu programa termina
  - › Array de bytes (palavra ByteArray)
  - › Array de caracteres (palavra CharArray)
  - › String (palavra String)
  - › Outro processo (palavra Piped)
- **Processamento intermediário realizado na informação**
  - › Bufferização (palavra Buffered)
    i) Na leitura, como o disco é lento, é melhor trazer muita informação de uma vez dentro de um lugar da memória (um array) que chamamos buffer
    ii)O acesso é feito diretamente à informação no buffer
    iii) Quando o buffer esvazia, é cheio novamente com uma única operação de leitura do disco
    iv) Idem para a saída
  - › Compressão (palavra ZIP)
    i) Para ler/escrever arquivo zip, por exemplo
  - › Para guardar o número da linha enquanto lê informação (palavra LineNumber)
  - › Para quebrar a informação em pedaços ou "tokens" (palavra Tokenizer)
- É por causa dessa grande variedade de opções que o sistema de entrada/saída (E/S ou do inglês I/O – input/output) do Java, baseado em streams, é meio chato de usar (mas não complicado)
  - › São muitas opções e é preciso conhecê-las para tomar boas decisões
  - › Ainda bem que podemos ter uma visão geral que nos dá um bom embasamento para essas decisões
  - › Nos deteremos mais aqui à persistência

Exemplo simples: leitura/escrita de bytes

- Todas as classes usadas para entrada/saída (leitura/escrita) de bytes de 8 bits são descendentes de

`InputStream` e `OutputStream`

- Vejamos um exemplo a seguir, que lê e escreve bytes de arquivos (FileInputStream e FileOutputStream)

```
package p2.exemplos;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
   public static void main(String[] args) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;
      try {
        in = new FileInputStream("proverbiochines.txt");
        out = new FileOutputStream("copiadeproverbiochines.txt");
        int c;

        while ((c = in.read()) != -1) {
           out.write(c);
        }

      } finally {
        if (in != null) {
           in.close();
        }
        if (out != null) {
           out.close();
        }
      }
   }
}
```

- É a stream mais primitiva

- Para ler caracteres, como no caso acima, seria mais indicado usar uma stream de caracteres e não de bytes
- As outras streams são construídas sobre as streams de bytes, por isso é importante conhecê-las
  - É como se pegássemos uma stream de bytes e a embrulhássemos (padrão *wrapper*) com uma capa que a torna uma stream de outro tipo de dado, como, por exemplo, caracteres
  - Usar o framework de I/O é basicamente um exercício de embrulhar da forma mais adequada uma stream de bytes
    i) Por isso vocês verão a seguir muito aninhamento de **`news`**…

Leitura/escrita de caracteres

- Também muito simples
- Todas as classes usadas para entrada/saída (leitura/escrita) de caracteres são descendentes de **`Reader`** e **`Writer`**
  - Note o padrão de nomes
    i) InputStream e OutputStream para bytes e Reader e Writer para caracteres
    ii)File para indicar que os dados vêm ou vão para um arquivo
- Vejamos um exemplo a seguir, que lê e escreve caracteres de arquivos (**`FileReader`** e **`FileWriter`**)

```
package p2.exemplos;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
   public static void main(String[] args) throws IOException {
      FileReader inputStream = null;
      FileWriter outputStream = null;
      int numeroDeCaracteres = 0;

      try {
         inputStream = new FileReader("proverbiochines.txt");
         outputStream = new FileWriter("copiacaracteres.txt");

         int c;
```

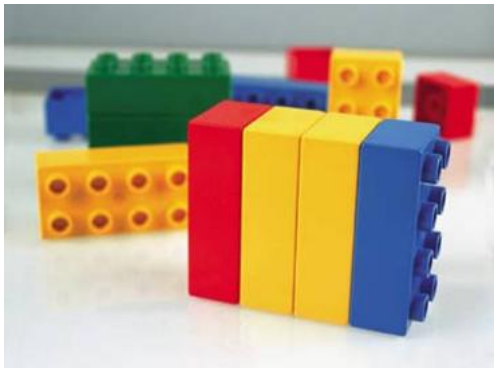
```
            while ((c = inputStream.read()) != -1) {
                numeroDeCaracteres++;
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
            System.out.println(numeroDeCaracteres +
                    " caracteres copiados para o arquivo copiacaracteres.txt.");
        }
    }
}
```

- Note que ambas usam uma variável inteira para ler/escrever em arquivos
  - A diferença é que no CopyBytes existe um byte nos 8 últimos bits do inteiro e no CopyCharacters existe um caractere nos 16 últimos bits do valor inteiro
  - FileReader só pode ser usado com arquivos de texto, enquanto FileInputStream adequa-se a arquivos de texto ou binário (arquivo de áudio, por exemplo)

Montagem de Sequências de Streams

- Quando queremos usar um stream, perguntamos primeiro:
  - **Em que direção a stream deve funcionar?**
  - **Qual é a fonte (na leitura) ou destino (na gravação) da informação?**
  - **Que tipo de informação será tratada?**
  - **Qual processamento(s) intermediário(s) deve(m) ser feito(s) na informação?**

- A beleza do sistema de E/S Java (e sua fonte de complexidade) é



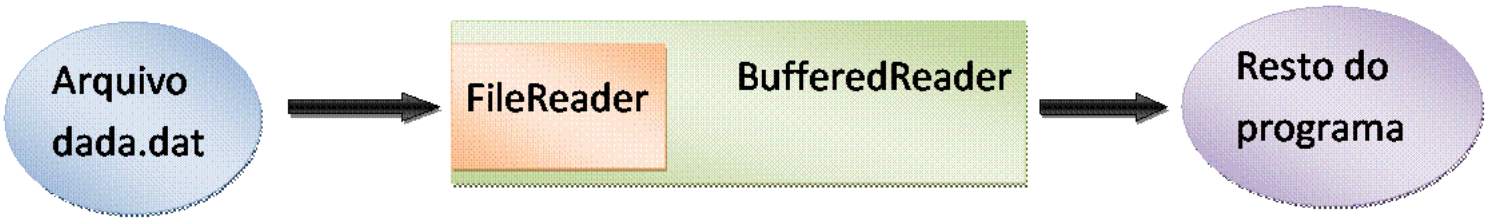
que podemos montar um stream juntando vários pedaços (como num jogo de Lego)

› Com algumas exceções, podemos tratar as 4 dimensões de forma independente, isto é, tomando decisões individuais para cada dimensão e juntando a solução para cada dimensão para fazer o stream final

- A montagem do stream é feita juntando objetos de várias classes através de um esquema de "*wrapping*" (empacotamento)
- Exemplo: Ler caracteres de um arquivo com “bufferização”
  - › Direção: Ler
  - › Fonte: Arquivo (File)
  - › Tipo: Caracteres (Reader)
  - › Processamento: Bufferização (Buffered)

```
BufferedReader in = new BufferedReader(
        new FileReader("dados.dat")
    );
```

- O stream que montamos é o seguinte:



	Lê	Grava	Bytes crus	Tipos nativos	Objetos	Carac-teres	Arquivo	Array de bytes	Outro processo	Array de carac-teres	String	Bufferi-zação	C

FileOutputStream		X	X	X	X		X						
ByteArrayOutputStream		X	X	X	X			X					
PipedOutputStream		X	X	X	X				X				
BufferedOutputStream		X	X	X	X							X	
ObjectOutputStream		X			X								
DataOutputStream		X		X									
ZIPOutputStream		X	X	X	X								X
GZIPOutputStream		X	X	X	X								X
FileInputStream	X		X	X	X		X						
ByteArrayInputStream	X		X	X	X			X					
PipedInputStream	X		X	X	X				X				
BufferedInputStream	X		X	X	X							X	
ObjectInputStream	X				X								
DataInputStream	X			X									
ZIPInputStream	X		X	X	X								X
GZIPInputStream	X		X	X	X								X
RandomAccessFile	X	X	X				X						
FileWriter		X				X	X						
PipedWriter		X				X			X				
CharArrayWriter		X				X				X			
StringWriter		X				X					X		
BufferedWriter		X				X						X	
PrintWriter (imprime formatado)		X				X							
FileReader	X					X	X						
PipedReader	X					X			X				
CharArrayReader	X					X				X			
StringReader	X					X					X		
BufferedReader	X					X						X	
LineNumberReader	X					X							
StreamTokenizer	X					X							

- Outro exemplo: queremos gravar dados em arquivo zipado com buferização
  - Direção: Escrever (Output)
  - Tipo: Bytes (Stream)
  - Fonte: Arquivo (File)
  - Processamento: Buferização (Buffered) + Compressão zip (Zip)

```
BufferedOutputStream out = new BufferedOutputStream(
        new ZipOutputStream(
            new FileOutputStream("dados.zip")
        )
    );
```

- Mais um exemplo: queremos gravar objetos em um arquivo

```
ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("agencia.txt"))
    );
```

Alguns Detalhes sobre o Conceito de Arquivo

- Arquivos são uma abstração que deu certo!
- Arquivos são usados devido à sua persistência
  - Isto é, o conteúdo não desaparece quando o programa termina
  - Isso não é o caso para objetos manipulados pelo programa
- Outro lugar para guardar informação persistente é um Banco de Dados
  - É uma "camada" em cima de arquivos para dar um acesso mais rápido quando há uma quantidade muito

grande de informação

- O acesso seqüencial proporcionado pelo arquivo seria muito lento
- O Banco de Dados fornece outros serviços, além de rapidez, tais como:

  i) Robustez

  ii)Agrupamento de operações (transações)

  iii) Controle de concorrência (quando vários usuários acessam os dados ao mesmo tempo)

  iv) etc.

- Ao usar "new FileReader(nomeArquivo)", o sistema operacional "abre" o arquivo
  - A "abertura" do arquivo tem vários propósitos:

    i) Verificar se o arquivo existe

    ii)Verificar se você tem permissão de usar o arquivo da maneira que está pedindo (ler ou gravar, por exemplo)

    iii) Preparar o sistema operacional para receber pedidos de acesso ao arquivo feitos pelo programa com a maior rapidez possível

      (1) Para atender a isso, o sistema operacional guarda informação do arquivo na memória para ter acesso mais rápido

- No final, é necessário avisar o sistema operacional que o arquivo não será mais usado, por enquanto
  - Isso é feito com a operação de fechamento de arquivo (close)

    i) O sistema só pode abrir um certo número limitado de arquivos, devido a limitações de recursos internos do sistema operacional (veja `ulimit` no Unix/Linux)

- Arquivos são freqüentemente organizados por registro
  - Exemplo: se um arquivo contiver o cadastro de alunos, um registro conteria o cadastro de um único aluno

- O registro consiste de campos (nome, matrícula, data de nascimento, ...)
- O registro pode ser de tamanho fixo (tamanho fixo para cada campo)
- O registro pode ser de tamanho variável (usando um separador especial entre campos)
- Às vezes, o arquivo pode ser organizado de outras formas e não conter apenas uma seqüência de registros do mesmo tipo

  i) Exemplos: arquivo HTML, arquivo XML, etc.
- É possível em muitas linguagens realizar acesso direto a registros específicos

java.io class RandomAccessFile

public void **seek**(long pos)

throws IOException

Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs. The offset may be set beyond the end of the file. Setting the offset beyond the end of the file does not change the file length. The file length will change only by writing after the offset has been set beyond the end of the file.

Parameters:

`pos` - the offset position, measured in bytes from the beginning of the file, at which to set the file pointer.

Throws:

`IOException` - if `pos` is less than `0` or if an I/O error occurs.

Mais um pouco sobre "Bufferização"

- O que é "bufferizar"?
  - Os exemplos mostrados não têm bufferização
- Por que "bufferizar"?

- Para reduzir custo, Java implementa streams de E/S “bufferizadas”
- Streams de entrada bufferizadas lêem dados de uma área de memória chamada “buffer”;
- A API nativa de Java para leitura em arquivo é chamada apenas quando o buffer está vazio;
- Streams de saída bufferizadas escrevem dados em uma área de memória chamada buffer; as chamadas nativas da API para efetivamente escrever no arquivo ocorrem apenas quando o buffer está cheio.

- Podemos converter streams não “bufferizadas” em streams bufferizadas
  - Empacotamento de streams
- Exitem 4 embrulhos para tornar streams não “bufferizadas” em streams “bufferizadas”
  - **BufferedInputStream** e **BufferedOutputStream** criam streams “bufferizadas” de bytes
  - **BufferedReader** e **BufferedWriter** criam streams “bufferizadas” de caracteres

Leitura/escrita de linhas inteiras

- Também muito simples
- É muito comum que a leitura/escrita de caracteres seja feita, de fato, considerando uma coleção de caracteres que forma uma linha
  - Uma linha é uma string de caracteres com um terminador de linha no fim
  - O finalizador de linha pode ser uma seqüência de carriage-return/line-feed (`"\r\n"`), a single carriage-return (`"\r"`), ou um simples line-feed (`"\n"`).
- É o caso em que há processamento; neste caso, bufferização
- Vamos modificar o exemplo **CopyCharacters** para aplicar I/O dirigida a linhas
  - Teremos que usar duas classes que não usamos ainda: **BufferedReader** e **PrintWriter**.

```
package p2.exemplos;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class CopyLines {
   public static void main(String[] args) throws IOException {
```

```
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader(
                    "proverbiochines.txt"));
            outputStream = new PrintWriter(new BufferedWriter(
                    new FileWriter("linesoutput.txt")));

            String line;
            while ((line = inputStream.readLine()) != null) {
                outputStream.println(line);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

- O readLine retorna uma linha com o finalizador de linha do sistema operacional onde a aplicação está sendo executada, mesmo que seja diferente do finalizador de linha encontrado no arquivo

Alguns Métodos Disponíveis para Manipular Streams

- (seção para estudar em casa!)
- As classes que definem streams em Java possuem vários métodos de acesso
  - Já usamos o método read() do BufferedReader, acima
- Resumimos aqui *alguns* dos métodos mais importantes de algumas classes
  - Veja a documentação da API Java para ver a lista completa de métodos

- InputStream
  - Todos os InputStreams podem:
    - i) Ler um byte: read()
    - ii)Ler vários bytes: read(byte[] b)
    - iii) Observe que a leitura de um byte retorna int
      - (1) Motivo: queremos retornar qualquer valor de byte e ainda ter o valor -1 para indicar o fim do stream
  - O ObjectInputStream ainda pode:
    - i) Ler um Objeto qualquer: readObject()
  - O DataInputStream ainda pode:
    - i) Ler dados de tipos básicos: readBoolean(), readDouble(), readInt(), ...
- OutputStream
  - Todos os OutputStreams podem:
    - i) Gravar um byte: write(int b)
    - ii)Gravar vários bytes: write(byte[] b)
  - O ObjectOutputStream ainda pode:
    - i) Gravar um Objeto qualquer: writeObject(Object obj)
  - O DataInputStream ainda pode:
    - i) Gravar dados de tipos básicos: writeBoolean(), writeDouble(), writeInt(), ...
- FileReader
  - Todos os FileReaders podem:

i) Ler um caractere: read()

(1) Retorna -1 no fim de arquivo

› O BufferedReader ainda pode:

i) Ler uma linha inteira: readLine()

› O LineNumberReader ainda pode:

i) Retornar o número da linha: getLineNumber()

- FileWriter
  - › Todos os FileWriters podem:
    - i) Gravar um caractere: write(int c)
  - › O BufferedWriter ainda pode:
    - i) Gravar um caractere de nova-linha (separador de linhas): newLine()
  - › O PrintWriter ainda pode:
    - i) Gravar dados de tipos básicos de forma formatada: print(int), print(double), ...

Streams de dados

- Apóia entrada/saída de tipos primitivos e strings
  - › `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, e `double`
- Na raiz dessas streams estão as interfaces DataInput e DataOutput
- Vejamos a seguir exemplos do uso de DataInputStream e DataOutputStream, que são implementações bastante usadas de DataInput e Dataoutput

```
package p2.exemplos;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
```

```
import java.io.DataOutputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class GravaRegistros {

   public static void main(String[] args) {
      String dataFile = "registros.txt";
      escreveDados(dataFile);
      leDados(dataFile);
   }

   private static void escreveDados(String dataFile) {
      double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
      int[] units = { 12, 8, 13, 29, 50 };
      String[] descs = { "Camiseta Java", "Caneca Java", "Boneco Java",
            "Broche Java", "Chaveiro Java " };

      DataOutputStream out = null;
      try {
         out = new DataOutputStream(new BufferedOutputStream(
               new FileOutputStream(dataFile)));

         for (int i = 0; i < prices.length; i++) {
            out.writeDouble(prices[i]);
            out.writeInt(units[i]);
            out.writeUTF(descs[i]);
         }

      } catch (FileNotFoundException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      } finally {
```

```
        if (out != null) {
            try {
                out.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

private static void leDados(String dataFile) {
    DataInputStream in = null;
    double price;
    int unit;
    String desc;
    double total = 0.0;
    try {
        in = new DataInputStream(new BufferedInputStream(
                new FileInputStream(dataFile)));
        try {
            while (true) {
                price = in.readDouble();
                unit = in.readInt();
                desc = in.readUTF();
                System.out.format("Voce pediu %d unidades de %s por $%.2f%n",
                        unit, desc, price);
                total += unit * price;
            }
        } catch (EOFException e) {
        } catch (IOException e) {
            e.printStackTrace();
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } finally {
        if (in != null) {
            try {
```

```
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    System.out.format("Valor total a pagar: $%.2f%n", total);
  }
}
```

- Observe que o programador tem que controlar o tipo de dado a ser lido
  - › Para cada write teve um read associado
- Observe o uso de EOFException para detectar o fim de arquivo

Streams de objetos

- A unidade que circula da stream é "objeto"
- As classes usadas são **ObjectInputStream** e **ObjectOutputStream**
  - › Implementam ObjectInput e ObjectOutput, que por sua vez herdam de DataInput e DataOutput respectivamente
    i) Assim, streams de objetos também são capazes de lidar com tipos primitivos e strings
- Programa idêntico ao anterior, mas que lida com quantias monetárias usando objetos da classe BigDecimal

```
package p2.exemplos;

import java.io.*;
import java.math.BigDecimal;
import java.util.Calendar;

public class ObjectStreams {
   static final String dataFile = "objetos.dat";

   static final BigDecimal[] prices = { new BigDecimal("19.99"),
         new BigDecimal("9.99"), new BigDecimal("15.99"),
         new BigDecimal("3.99"), new BigDecimal("4.99") };
```

```
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = { "Camiseta Java", "Caneca Java",
      "Boneco Java", "Broche Java", "Chaveiro Java" };

public static void main(String[] args) throws IOException,
      ClassNotFoundException {

   writeObjects();
   readObjects();
}

private static void writeObjects() throws IOException,
      FileNotFoundException {
   ObjectOutputStream out = null;
   try {
      out = new ObjectOutputStream(new BufferedOutputStream(
            new FileOutputStream(dataFile)));

      out.writeObject(Calendar.getInstance());
      for (int i = 0; i < prices.length; i++) {
         out.writeObject(prices[i]);
         out.writeInt(units[i]);
         out.writeUTF(descs[i]);
      }
   } finally {
      out.close();
   }
}

private static void readObjects() throws IOException,
      FileNotFoundException, ClassNotFoundException {
   ObjectInputStream in = null;
   try {
      in = new ObjectInputStream(new BufferedInputStream(
            new FileInputStream(dataFile)));

      Calendar date = null;
```

```
            BigDecimal price;
            int unit;
            String desc;
            BigDecimal total = new BigDecimal(0);

            date = (Calendar) in.readObject();

            System.out.format("%tA, %<tB %<te, %<tY:%n", date);

            try {
               while (true) {
                  price = (BigDecimal) in.readObject();
                  unit = in.readInt();
                  desc = in.readUTF();
                  System.out.format("Você pediu %d unidades de %s por $%.2f%n",
                                    unit, desc, price);
                  total = total.add(price.multiply(new BigDecimal(unit)));
               }
            } catch (EOFException e) {
            }
            System.out.format("Valor TOTAL da compra: $%.2f%n", total);
         } finally {
            in.close();
         }
      }
}
```

- Se um objeto tiver atributos que são referências a outros objetos, então uma escrita de um objeto vai resultar, de fato, na escrita de vários objetos
  - E se dois objetos referenciarem o mesmo objeto?

Serialização

- Anda lado a lado com streams de objetos que empacotam streams de dados, que empacotam streams de bytes!
- A serialização significa gravar, seqüencialmente e byte-por-byte, os atributos de um objeto
- Qual é o tempo máximo de vida de um objeto?

- Que tal ter objetos que ficam em um estado dormente, mas com todo o seu estado mantido, mesmo depois que o programa termina?
- Na próxima vez que o programa for executado, o objeto terá a mesma informação/estado que ele tinha quando o programa foi executado pela última vez
- Surgiu para permitir: RMI (Remote Method Invovation) e JavaBeans

- Como tornar um objeto persistente?
  - Implementando a interface Serializable
    i) É uma interface de rotulação ("*tagging*") e não contém métodos a serem implementados
  - O objeto pode ser transformado em uma seqüência de bytes (que será armazenado em um arquivo) e que pode depois ter seu estado restaurado
  - O mecanismo de serialização está preparado para lidar com diferenças de representação entre diferentes sistemas
- É preciso "serializar" e "desserializar" os objetos explicitamente gravando-os em arquivo como vimos no exemplo anterior
- Para serializar criamos um ObjectOutputStream que empacota um OutputStream e chamamos o método writeObject()
- Para desserializar fazemos o processo inverso: criamos um objeto ObjectInputStream que empacota um OutputStream e usamos readObject()
  - Deve haver o cuidado ao fazer o downcast…
- Vejamos um exemplo

```
package p2.exemplos;

import java.io.*;
import java.math.BigDecimal;

public class Serializavel implements Serializable {

   private static final long serialVersionUID = 1L;
   private Serializavel instancia;
   private String string;
   private BigDecimal bigDecimal;
```

```
public Serializavel(int numInstances) {
   bigDecimal = new BigDecimal(numInstances);
   string = "serializavel numero " + numInstances;
   if (--numInstances > 0)
      instancia = new Serializavel(numInstances);
   else
      instancia = null;
}

public Serializavel getOtherInstance() {
   return instancia;
}

public Serializavel getInstance() {
   return this;
}

public String getString() {
   return string;
}

public BigDecimal getBigDecimal() {
   return bigDecimal;
}

@Override
public String toString() {
   if (instancia != null)
      return string + "; " + bigDecimal.toPlainString() + "; "
            + instancia;
   return string + "; " + bigDecimal.toPlainString() + ";";
}

public static void main(String[] args) throws FileNotFoundException,
      IOException {

   int profundidade = 6;
```

```
        Serializavel serializavel = new Serializavel(profundidade);

        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(
                "serializavel.ser"));
        out.writeObject(serializavel);
        out.close();
    }
}
```

```
package p2.exemplos;

import java.io.*;

public class LeSerializavel {
public static void main(String[] args) throws FileNotFoundException,
IOException, ClassNotFoundException {

        ObjectInputStream in = new ObjectInputStream(new
FileInputStream("serializavel.ser"));
        Serializavel serializavel = (Serializavel) in.readObject();
        System.out.println(serializavel);
        in.close();
    }
}
```

Outras questões mais avançadas

- Controle de versões usando ***serialVersionUID***
- Atributos transientes
  - Palavra chave `transient`
- A interface Externalizable que herda de Serializable
  - writeExternal()
  - readExternal()

- http://java.sun.com/javase/6/docs/platform/serialization/spec/version.html
- http://blog.caelum.com.br/2008/04/01/entendendo-o-serialversionuid/
- http://www.javabeat.net/tips/11-using-the-externalizable-interface.html

Voltar