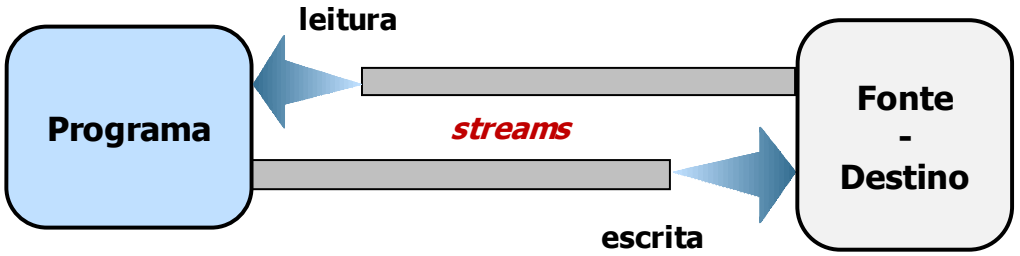
STREAMS DE JAVA

Em JAVA, todas as considerações (ou quase todas) que se relacionam com as mais diferentes formas de se realizar a LEITURA e a ESCRITA DE DADOS a partir das mais diversas fontes e para os mais diferentes destinos, são reunidas e abstraídas no conceito de *STREAM*.

Uma *STREAM* é uma abstracção que representa uma fonte genérica de entrada de dados ou um destino genérico para escrita de dados, de acesso sequencial e independente de dispositivos físicos concretos, formatos ou até de mecanismos de optimização de leitura e escrita. É, portanto, UMA ABSTRACÇÃO e, como tal, terá que ser sempre refinada e concretizada, e, em particular, ser ASSOCIADA A UMA ENTIDADE FÍSICA DE SUPORTE DE DADOS, seja um ficheiro em disco ou em CD-ROM, um *WEBSITE*, um *ARRAY* de *BYTES*, uma *STRING*, um DVD, um outro computador da rede, etc.



***Streams* como abstracções de leitura e escrita**

Para ler informação, um programa abre uma *STREAM* sobre uma dada fonte de informação, por exemplo um ficheiro, a memória, um *SOCKET*, e lê essa informação sequencialmente, ou *BYTE* a *BYTE* ou carácter a carácter. Inversamente, um programa pode enviar informação para um destino externo abrindo uma *STREAM* de escrita e escrevendo informação de modo sequencial na *STREAM*.

AS OPERAÇÕES DE LEITURA E ESCRITA SOBRE UMA QUALQUER *STREAM*, OBEDECEM A UM PADRÃO DE FUNCIONAMENTO E UTILIZAÇÃO QUE É SEMPRE MUITO SEMELHANTE.

```
try {
   abrir a stream
   ler
   enquanto não for fim
      processar
      ler
}
catch(IOException e) { .. }
fechar a stream
```

```
try {
      abrir a stream
   processar informação
      enquanto não for fim
      escrever na stream
      processar informação
}
      catch(IOException e) { .. }
fechar a stream
```

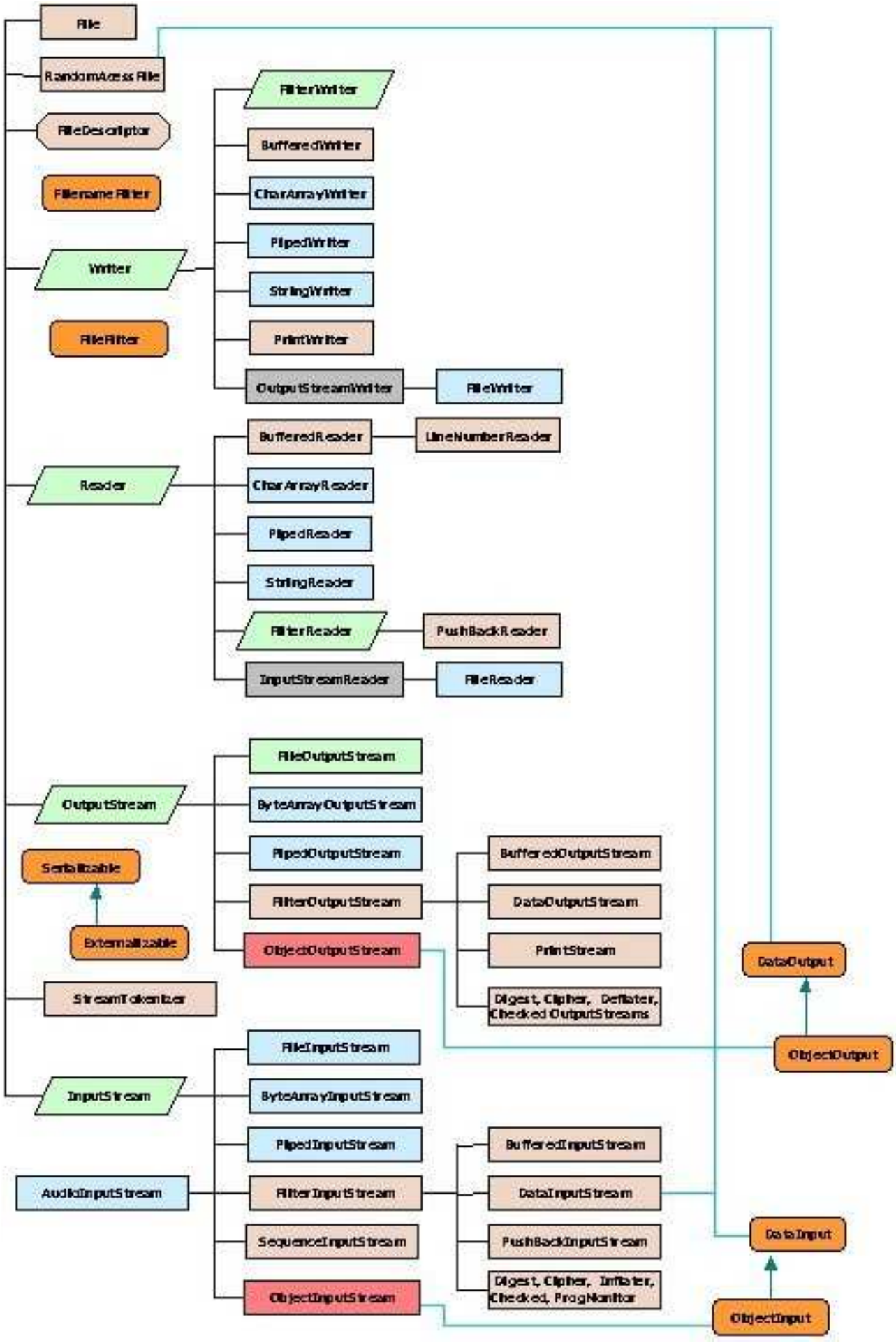
Em JAVA, existem dois grandes tipos de **streams**, designadamente,

- ***Streams* de caracteres (2 bytes)**, ou seja, *streams* de texto;
- ***Streams* de bytes**, ou seja, *streams* binárias.

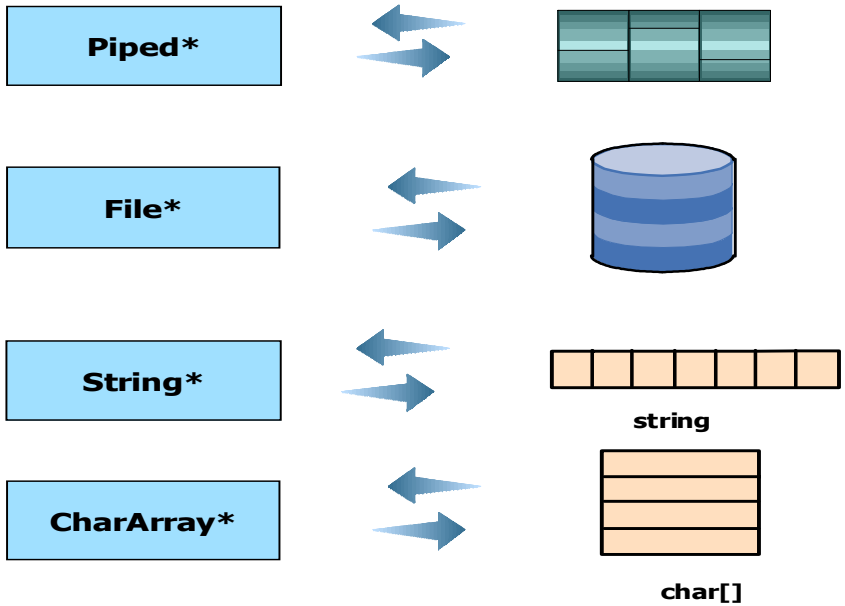
com dois tipos de funções fundamentais:

- **Leituras - Readers**
- **Escritas** - **Writers**

VEJAMOS A HIERARQUIA DE STREAMS DE JAVA:


File
RandomAccessFile
FileDescriptor
FilenameFilter
Writer
FileFilter
FilterWriter
BufferedWriter
CharArrayWriter
PipedWriter
StringWriter
PrintWriter
OutputStreamWriter
FileWriter
Reader
BufferedReader
LineNumberReader
CharArrayReader
PipedReader
StringReader
FilterReader
PushBackReader
InputStreamReader
FileReader
OutputStream
FileOutputStream
ByteArrayOutputStream
PipedOutputStream
FilterOutputStream
ObjectOutputStream
BufferedOutputStream
DataOutputStream
PrintStream
Digest, Cipher, Deflater, CheckedOutputStreams
Serializable
Externalizable
StreamTokenizer
InputStream
FileInputStream
ByteArrayInputStream
PipedInputStream
FilterInputStream
SequenceInputStream
ObjectInputStream
BufferedInputStream
DataInputStream
PushBackInputStream
Digest, Cipher, Inflater, CheckedProgMonitor
AudioInputStream
DataOutput
ObjectOutput
DataInput
ObjectInput

Algumas Streams foram concebidas para comunicarem directamente com fontes ou destinos de dados.



As diversas subclasses de Writer, Reader, InputStream e OutputStream, possuem designações iniciadas por prefixos, tais como Print, Buffered, File, Piped, CharArray, String, etc., que já analisámos atrás.

Temos agora, na prática que saber tomar a decisão sobre qual a ***stream lógica*** que nos interessa considerar ao nível da programação, ou seja, a classe que possui os métodos que pretendemos utilizar, e a ***stream física*** que pretendemos usar por ser a mais adequada em termos de fonte ou destino dos dados.

ANINHAMENTO DE STREAMS

A maioria dos construtores de Streams apresenta uma declaração "aninhada" que pode parecer estranha, mas que reflecte esta distinção entre stream lógica e stream física.

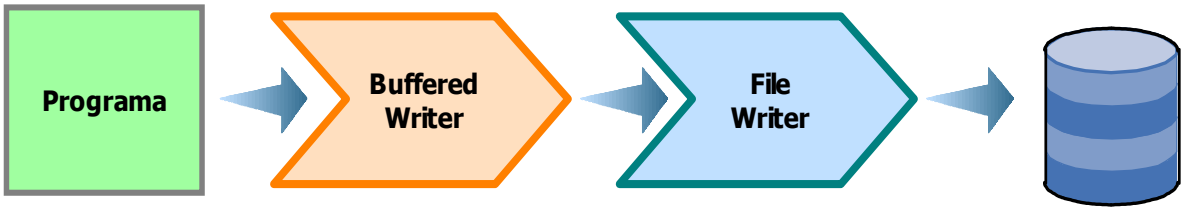
```
public BufferedWriter(Writer out);
```

Exemplo:

```
BufferedWriter bout = new BufferedWriter( new FileWriter("f1.txt") );
```

O que se consegue com este tipo de aninhamento é, antes de mais, a possibilidade de, ao nível da programação, usarmos um protocolo ou API de alto nível, que nos permite abstrair de um grande número de detalhes concretos de implementação sobre a obtenção dos dados na fonte (neste caso, o programa) e a sua transmissão efectiva para o destino, neste caso, um ficheiro de caracteres (onde cada carácter é representado usando 2 *bytes*).

A Figura procura ilustrar, em concreto, esta situação de **aninhamento**, ou associação (*wrapping*, segundo alguns), entre *streams* lógicas e físicas:



A simetria da Hierarquia é uma enorme vantagem:

```
BufferedReader bin = new BufferedReader( new FileReader("f1.txt") );
```

A NOVA VERSÃO (DE JAVA5) DE `PrintWriter` PERMITE QUE ESTA SEJA USADA DIRECTAMENTE SOBRE UM FICHEIRO EM DISCO. VEJAMOS A SUA EFICIÊNCIA:

```
try {
   PrintWriter pw = new PrintWriter("fichp52.txt");
   for(Produto pr : stock.values()) pw.print(pr);                    // usa toString()
   pw.flush(); pw.close();
}
catch(IOException e) { out.println(e.getMessage()); }
```

A Tabela sintetiza os resultados obtidos (100.000 fichas de produtos).

***Stream* usada**	***Stream* de Interface**	**Tempo Médio**
BufferedWriter	FileWriter (c/ toString())	1090 ms
FileWriter	----------	1180 ms
PrintWriter	FileWriter (c/ toString())	1200 ms
PrintWriter	FileWriter (por campos)	1440 ms
PrintWriter	BufferedWriter, FileWriter (c/ toString())	1080 ms
PrintWriter	BufferedWriter, FileWriter (por campos)	780 ms
PrintWriter	FileOutputStream (c/ toString())	1100 ms
PrintWriter	FileOutputStream (por campos)	790 ms
PrintWriter	----------	820 ms !!

Tabela global comparativa de tempos de gravação

CLASSES DE INTERESSE IMEDIATO PARA LEITURA:

- BufferedReader + FileReader
- Scanner
- ObjectInputStream
- DataInputStream (complexas !!)

CLASSES DE INTERESSE IMEDIATO PARA ESCRITA:

- PrintWriter
- BufferedWriter + FileWriter
- ObjectOutputStream
- DataOutputStream (complexas !!)

EXEMPLO SIMPLES: LER LINHAS DE UM FICHEIRO PARA UM ARRAYLIST<STRING>

```
import java.io.*;
public class TextReader {
  private static ArrayList<String> readFile(String fileName) {
    ArrayList<String> linhas = new ArrayList<String>();
    try {
      FileReader reader = new FileReader(fileName);
      BufferedReader in = new BufferedReader(reader);
      String string;
      while ((string = in.readLine()) != null) {
        linhas.add(string);
      }
      in.close();
    } catch (IOException e) { e.printStackTrace(); }
  return linhas;
  }

  public static void main(String[] args) {
    ArrayList<String> lines = new ArrayList<String>();
    if (args.length != 1) {
      System.out.println("Erro no ficheiro parâmetro !!");
      System.exit(0);
    }
    lines = readFile(args[0]);
   ……….
  }
}
```

EXEMPLO SIMPLES: LER LINHAS DE PALAVRAS DE UM FICHEIRO E IMPRIMIR AS PALAVRAS, UMA POR LINHA, NO MONITOR (USO DE SCANNER PARA LEITURA DE FICHEIRO).

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class TextScanner {

  private static void readFile(String fileName) {
    try {
      Scanner scanner = new Scanner( new File(filename) );
      while (scanner.hasNext()) {
        System.out.println(scanner.next());
      }
      scanner.close();
    } catch (FileNotFoundException e) { e.printStackTrace(); }
  }

  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("Erro no ficheiro parâmetro !!");
      System.exit(0);
    }
    readFile(args[0]);
  }
}
```

EXEMPLO SIMPLES: CONTAGEM DO NÚMERO DE LINHAS DE UM FICHEIRO DE TEXTO

```
// Contagem do Total de Linhas de um ficheiro
String linha = ""; int contaL = 0;
try {
    BufferedReader bin = new BufferedReader(new FileReader("fichp1.txt");
    while ( (linha = bin.readLine()) != null ) {
      contaL++;
    }
    bin.close();
}
catch(IOException e){ e.getMessage(); }
out.println("Total de linhas = " + contaL);
```

OU AINDA, USANDO O MÉTODO READY() QUE TESTA SE AINDA HÁ CARACTERES PARA LER,

```
while(bin.ready()) {
  linha = bin.readLine();
  contaL++;
}
```

Problema Tipo De Utilização das Classes Fundamentais sobre Ficheiros:

Criar uma classe Stock de Produto. Gravar um Stock em Ficheiro de texto, 1 produto por linha e com "," como separador. Posteriormente, ler tal ficheiro de texto e criar o estado de uma nova instância de stock.

```
/**
 * Produto:
* @author F. Mário Martins
*/
import java.io.Serializable;
public class Produto implements Serializable {
        // variáveis de instância
        private String codigo;
        private String nome;
        private int quant;
        private double preco;
        // Construtores
        public Produto() {
           codigo = "?"; nome = "?"; quant = 0; preco = 0.0;
        }
        public Produto(String cod, String nm, int qt, double pre) {
           codigo = cod; nome = nm; quant = qt; preco = pre;
        }
        public Produto(Produto p) {
            codigo = p.getCodigo(); nome = p.getNome();
            quant = p.getQuant(); preco = p.getPreco();
        }
```

```
    // Métodos de Instância
    public String getCodigo() { return codigo; }
    public String getNome() { return nome; }
    public void setNome(String nvNome) { nome = nvNome; }
    public int getQuant() { return quant; }
    public void mudaQuant(int val) { quant += val; }
    public double getPreco() { return preco; }
    public void mudaPreco(double nvPreco) { preco = nvPreco; }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if ((obj == null) || (this.getClass() != obj.getClass())) return false;
        Produto p = (Produto) obj;  // casting para tipo X
        return this.getCodigo().equals(p.getCodigo()) && this.getNome().equals(p.getNome()) &&
                this.getQuant() == p.getQuant() && this.getPreco() == p.getPreco();
    }

    public String toString() {
      StringBuilder sb = new StringBuilder("-------------------\n");
      sb.append("Codigo: " + codigo + "\n");
      sb.append("Nome: " + nome + "\n");
      sb.append("Quant: " + quant + "\n");
      sb.append("Preço: " + preco + "\n");
      sb.append("---------------------\n");
      return sb.toString();
    }
    public Produto clone() { return new Produto(this); }
}
```

```
/**
 * Stock: gestão de um Stock de produtos
 *
 * @author F. Mário Martins
 * @version 1/2006
 */
import java.io.Serializable;
import java.util.*;
import java.io.*;
public class Stock implements Serializable {
        //
        private TreeMap<String, Produto> stock;

        // Construtores
        public Stock() { stock = new TreeMap<String, Produto>(); }

        public Stock(TreeMap<String, Produto> stk) {
            stock = new TreeMap<String, Produto>();
            for(Produto p : stk.values())
               stock.put(p.getCodigo(), p.clone());
        }

        public Stock(Stock stk) {
            TreeMap<String, Produto> aux = stk.getStock();
            for(Produto p : aux.values())
               stock.put(p.getCodigo(), p.clone());
        }
```

```
// Métodos de instância
public TreeMap<String, Produto> getStock() {
    TreeMap<String, Produto> aux = new TreeMap<String, Produto>();
    for(Produto p : stock.values())
        aux.put(p.getCodigo(), p.clone());
    return aux;
}

// Devolve o conjunto dos códigos
public Set<String> codigos() {
    return stock.keySet();
}

// Devolve a lista dos produtos em stock
public List<Produto> produtos() {
    ArrayList<Produto> prods = new ArrayList<Produto>();
    for(Produto p : stock.values()) prods.add(p.clone());
    return prods;
}

// insere um novo produto
public void insereProduto(Produto p) {
    stock.put(p.getCodigo(), p.clone());
}

public void removeProd(String cod) {
    stock.remove(cod);
}
```

```
    // dá a ficha de um produto de código dado
    public Produto getProduto(String cod) {
       return stock.get(cod).clone();
    }

    // incrementa a quantidade em stock
    public void aumentaQuantProd(String cod, int qt) {
       stock.get(cod).mudaQuant(qt);
    }

    public String toString() {
       StringBuilder sb = new StringBuilder("------- STOCK --------\n");
       for(Produto p : stock.values()) sb.append(p.toString());
          sb.append("-------------------------\n");
    return sb.toString();
}

    public Stock clone() {
       return new Stock(this);
    }

    // grava o stock actual num ficheiro de objectos
    public void gravaObj(String fich) throws IOException {
          ObjectOutputStream oos = new ObjectOutputStream( new FileOutputStream(fich));
          oos.writeObject(this);
          oos.flush(); oos.close();
    }
```

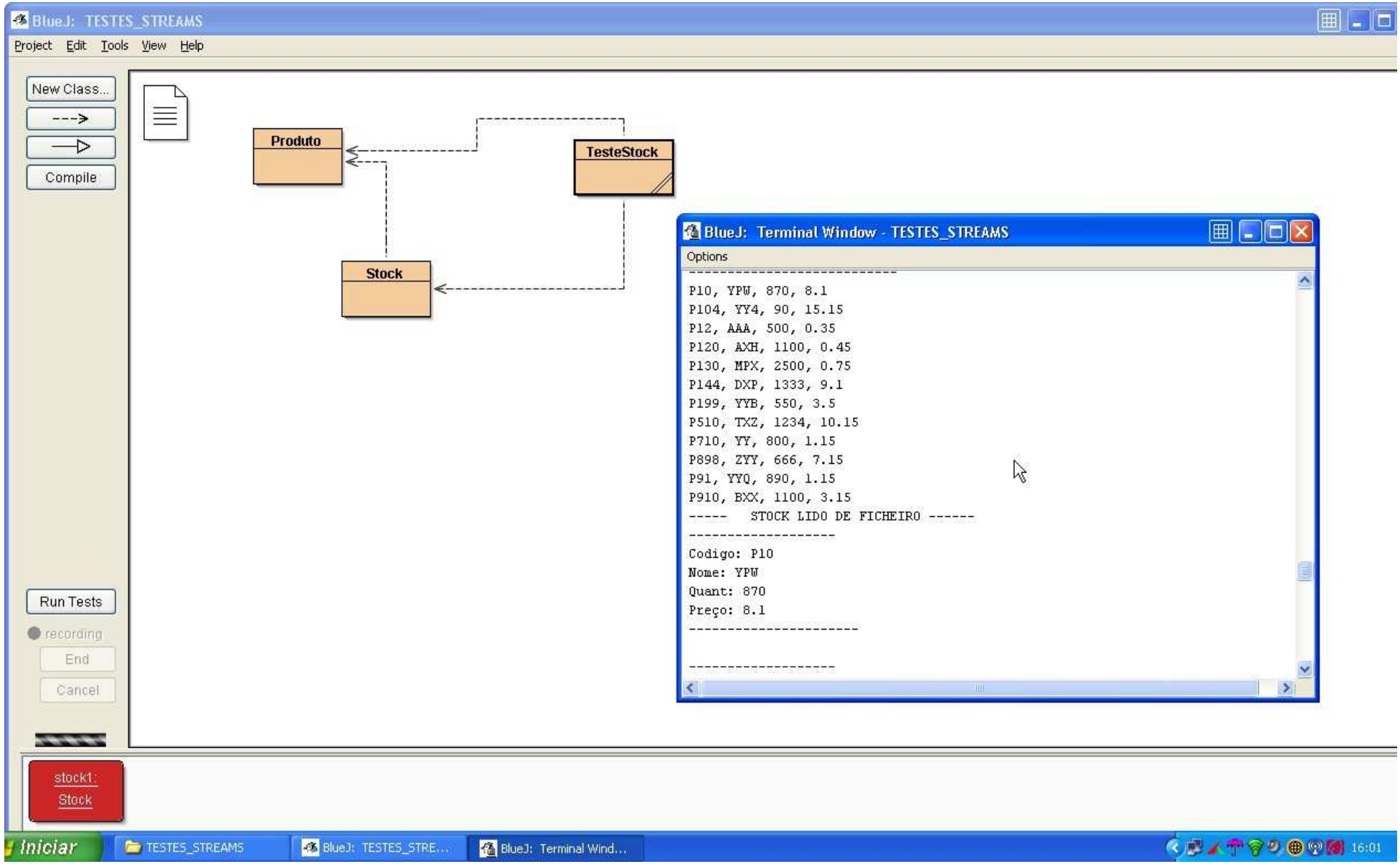
```
   // grava o stock num ficheiro de texto
   public void gravaTxt(String fich) throws IOException {
      PrintWriter pw = new PrintWriter(fich);
      pw.print(this);
      pw.flush(); pw.close();
   }

}
```

```
/**
 * TesteStock.
 *
 * Esta classe de Teste de Stock, para além de criar um stock, mostra como se podem gravar
 * registos de produtos em ficheiro de texto usando PrintWriter e mostra também como usando
 * Scanner tais registos podem ser lidos do ficheiro de texto criado de modo a recriar o stock a
 * partir de ficheiro de texto.
 *
* Nota: Os valores double são escritos em ficheiro de forma normal (cf. 37.45). Porém, o método
 * nextDouble() da classe Scanner lê valores double como sendo 37,45, ou seja, tendo a vírgula
 * como separador. Assim, para os double (reais), deve ler-se uma String e convertê-la para double.
 *
 * @author F. Mário Martins
 * @version 1/2006
 */
import java.io.*;
import java.util.*;
import static java.lang.System.out;
public class TesteStock {

   public static Stock criaStock() {
        Stock stk = new Stock();
       Produto p = new Produto("P12", "AAA", 500, 0.35); stk.insereProduto(p);
       p = new Produto("P10", "AXX", 1500, 0.15); stk.insereProduto(p);
       p = new Produto("P91", "YYQ", 890, 1.15); stk.insereProduto(p);
       p = new Produto("P120", "AXH", 1100, 0.45); stk.insereProduto(p);
       p = new Produto("P710", "YY", 800, 1.15); stk.insereProduto(p);
```

```
    p = new Produto("P130", "MPX", 2500, 0.75); stk.insereProduto(p);
    p = new Produto("P104", "YY4", 90, 15.15); stk.insereProduto(p);
    p = new Produto("P510", "TXZ", 1234, 10.15); stk.insereProduto(p);
    p = new Produto("P199", "YYB", 550, 3.5); stk.insereProduto(p);
    p = new Produto("P910", "BXX", 1100, 3.15); stk.insereProduto(p);
    p = new Produto("P10", "YPW", 870, 8.1); stk.insereProduto(p);
    p = new Produto("P144", "DXP", 1333, 9.1); stk.insereProduto(p);
    p = new Produto("P898", "ZYY", 666, 7.15); stk.insereProduto(p);
    return stk;
}

public static void gravaFichTxt(Stock stk, String fich) throws IOException {
    PrintWriter pw = new PrintWriter(fich);
    for(Produto p : stk.getStock().values()) {
        pw.print(p.getCodigo() + ", ");
        pw.print(p.getNome() + ", ");
        pw.print(p.getQuant() + ", ");
        pw.println(p.getPreco() + " ");
     }
     pw.flush(); pw.close();
 }

public static Stock leFichProdutos(String fich) {
    Stock stk = new Stock(); Produto p = null;
    Scanner scan = null; String linha = null;
    try {
        scan = new Scanner(new FileReader("stock2.txt"));
```

```
            // deve-se usar o separador de linhas default da JVM
            scan.useDelimiter(System.getProperty("line.separator"));
            while(scan.hasNext()) {
                linha = scan.next();
                out.println(linha);
                p = parseLinha(linha);
                stk.insereProduto(p); p = null;
            }
        }
        catch(IOException ioExc) { out.println(ioExc.getMessage()); }
        finally { scan.close(); }   // este bloco é sempre executado, haja erro ou não !!
        return stk;
    }

    public static Produto parseLinha(String linha) {
        String codigo, nome; int quant = 0; double preco = 0.0;
        Scanner lineScan = new Scanner(linha);
        lineScan.useDelimiter("\\s*,\\s*");
        codigo = lineScan.next(); nome = lineScan.next();
        quant = lineScan.nextInt();
        // preco = lineScan.nextDouble(); -> ERRO espera ,
        // temos que converte String -> Double -> double
        preco = Double.valueOf(lineScan.next()).doubleValue();

        return new Produto(codigo, nome, quant, preco);
    }
```

```
public static Stock main() {
    // inicializa o stock
    Stock stock = criaStock();
    // verifica qual o stock criado
    out.println("-----   STOCK CRIADO ------ ");
    TreeMap<String, Produto> prods = stock.getStock();
    for(Produto p : prods.values()) out.println(p.toString());
    out.println("-------------------------- ");
    // grava o stock num ficheiro de texto. 1 linha = 1 produto
    try {
        gravaFichTxt(stock, "stock2.txt");
    }
    catch(IOException e) { out.println(e.getMessage()); }
    // recria o stock a partir do ficheiro de texto
    // stock = null garante que não é o que já existia
    stock = null; stock = leFichProdutos("stock2.txt");

    // verifica qual o stock lido
    out.println("-----   STOCK LIDO DE FICHEIRO ------ ");
    prods = null; prods = stock.getStock();
    for(Produto p : prods.values()) out.println(p.toString());
    out.println("-------------------------- ");

    return stock;
  }
}
```


BlueJ: TESTES_STREAMS
Project Edit Tools View Help
New Class...
Compile
Run Tests
recording
End
Cancel
Produto
TesteStock
Stock
stock1:
Stock
BlueJ: Terminal Window - TESTES_STREAMS
Options
P10, YPW, 870, 8.1
P104, YY4, 90, 15.15
P12, AAA, 500, 0.35
P120, AXH, 1100, 0.45
P130, MPX, 2500, 0.75
P144, DXP, 1333, 9.1
P199, YYB, 550, 3.5
P510, TXZ, 1234, 10.15
P710, YY, 800, 1.15
P898, ZYY, 666, 7.15
P91, YYQ, 890, 1.15
P910, BXX, 1100, 3.15
----- STOCK LIDO DE FICHEIRO ------
Codigo: P10
Nome: YPW
Quant: 870
Preço: 8.1
Iniciar
TESTES_STREAMS
BlueJ: TESTES_STRE...
BlueJ: Terminal Wind...
16:01

OBJECT STREAMS

- **Permitem guardar e ler de ficheiro instâncias de classes Serializable;**
- **São muito eficientes e muito simples de usar**

```
// Grava a variável de instância stock (TreeMap<String, Produto>) numa ObjectStream
try {
  ObjectOutputStream oout =
    new ObjectOutputStream(new FileOutputStream("Stock.dat"));
  oout.writeObject(stock);
  oout.flush(); oout.close();
}
catch(IOException e) { System.out.println(e.getMessage()); }
```

```
// Lê o TreeMap<String, Produto>
Stock stock = null;
try {
  ObjectInputStream oin =
    new ObjectInputStream(new FileInputStream("Stock.dat"));
  stockNovo = (TreeMap<String,Produto>) oin.readObject();
  oin.close();
}
catch(IOException e) { System.out.println(e.getMessage()); }
catch(ClassNotFoundException e) { System.out.println(e.getMessage()); }
// reconstrói
stock = new Stock(stockNovo);
```

ou ainda

```
Stock mystock = new Stock();
.....

// Grava a instância de Stock numa ObjectStream
try {
  ObjectOutputStream oout =
    new ObjectOutputStream(new FileOutputStream("MyStock.dat"));
  oout.writeObject(myStock);
  oout.flush(); oout.close();
}
catch(IOException e) { System.out.println(e.getMessage()); }
```

```
// Lê o Stock gravado para um novo stock
Stock stockNovo = null;
try {
  ObjectInputStream oin =
    new ObjectInputStream(new FileInputStream("MyStock.dat"));
  stockNovo = (Stock) oin.readObject();
  oin.close();
}
catch(IOException e) { System.out.println(e.getMessage()); }
catch(ClassNotFoundException e) { System.out.println(e.getMessage()); }
```