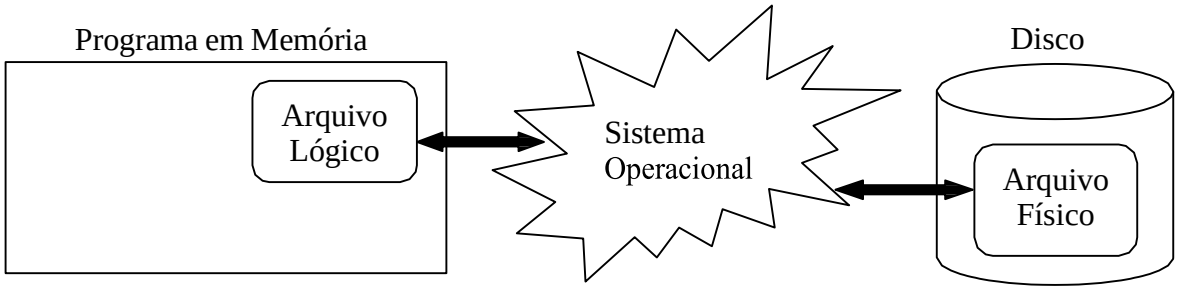
# CONCEITOS BÁSICOS E MANIPULAÇÃO DE ARQUIVOS EM JAVA

## 1 Conceitos Básicos

### *1.1 Arquivos Físicos × Arquivos Lógicos*

Arquivos Físicos representam um conjunto de bytes armazenado em um meio físico. São administrados/manipulados, via de regra, pelo Sistema Operacional (SO).

Arquivos Lógicos são estruturas de dados de um programa que representam um arquivo físico, isto é, permitem acesso/manipulação do arquivo pelo programa. A iteração entre um programa (arquivo lógico) e um arquivo físico é mediada pelo SO.




### *1.2 Tipos de Arquivos*

Os arquivos podem ser agrupados em 3 categorias (ou tipos) :

a) Arquivos Regulares : Contendo dados/informações e programas. Podem ser subdivididos em :
   1. Binário - Sequência de Bytes
   2. Texto - Sequência de Caracteres

b) Arquivos de Sistema : Contendo dados do SO (diretórios)

c) Arquivos Especiais (ou de Dispositivos)

### *1.3 Modos de Acesso*

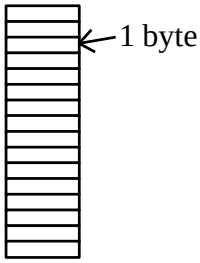
Os arquivos podem ser acessados de duas formas :

- Acesso Sequencial
- Acesso Randômico (ou Aleatório)

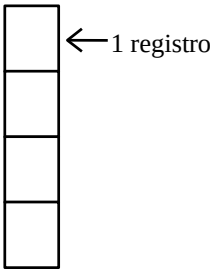
## *1.4 Estruturação*

Os arquivos podem estar organizados (internamente) de várias formas :

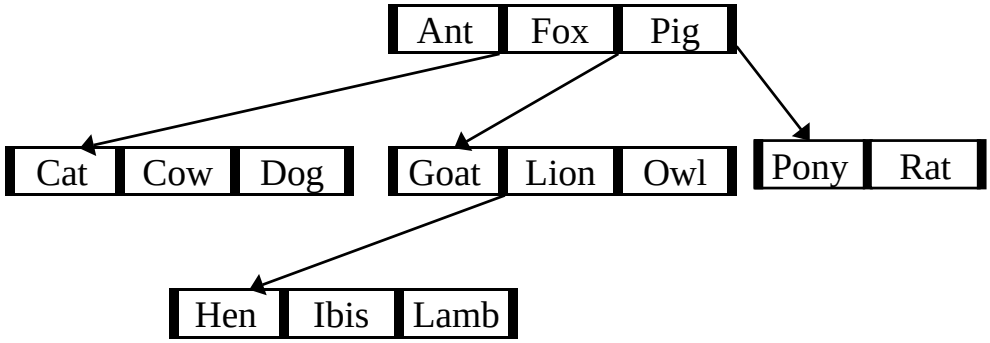
- Sequência não estruturada de bytes
- Sequência de registros (conjunto de bytes)
- Estrutura de pesquisa (por exemplo: árvore) contendo registros




Seqüência não estruturada de bytes




Seqüência de registros



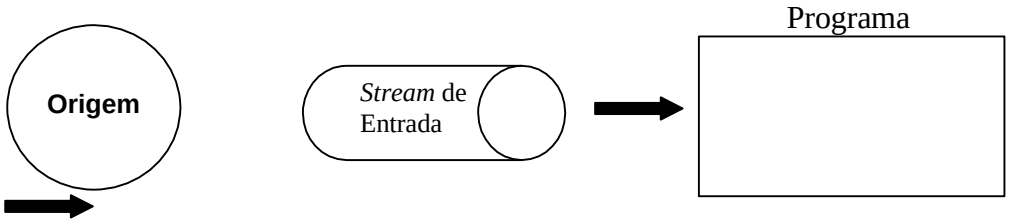

Árvore de registros

# 2 Manipulação de Arquivos em Java

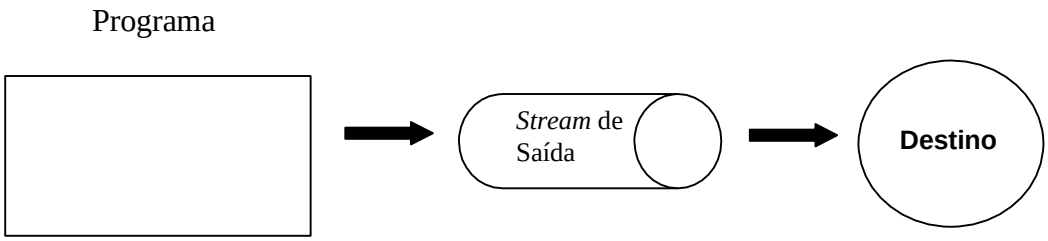
## *2.1 Streams em Java*

*Streams* representam mecanismos para envio e recepção de informação em programas em Java. A operação de ESCRITA é responsável pelo envio de dados; enquanto a operação de LEITURA é responsável recepção de dados.

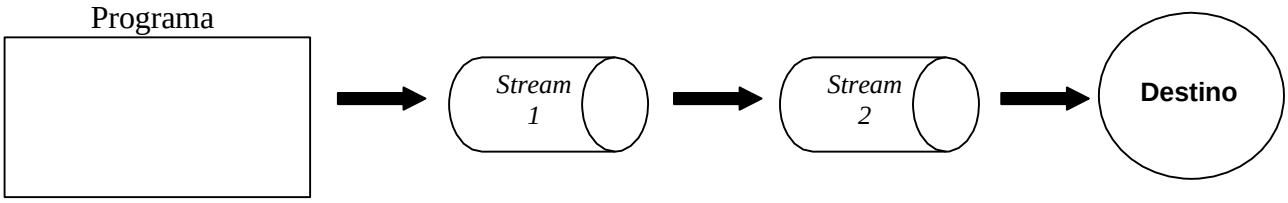
***Stream* de Entrada (*InputStream*)**




***Stream* de Saída (*OutputStream*)**




Os *streams* podem ser concatenados de modo a permitir a implementação de filtros e/ou serviços especializados (como, por exemplo, o leitura ou armazenamento de objetos).

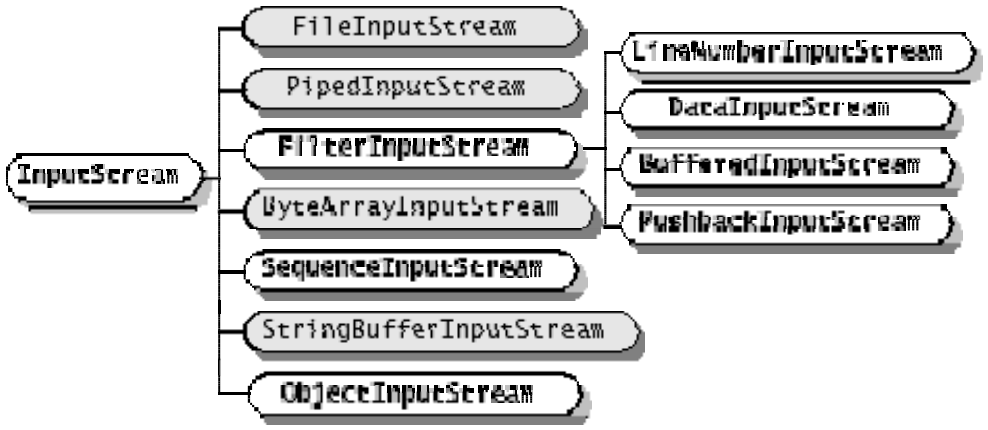



O pacote **java.io** contém uma série de classes que podem ser usadas para manipulação de *streams*. Um *stream* representa um fluxo de bytes unidirecional. Sendo assim, um *stream* de entrada só pode ser usado para operações de leitura; enquanto que um *stream* de saída só pode ser utilizado para operações de escrita. Isto pode eventualmente representar uma limitação (caso se queira ler e escrever ao mesmo tempo) e esse obstáculo poderia ser contornado utilizando-se da classe **RandomAccessFile**.

## 2.2 Byte Streams

O pacote **java.io** define classes abstratas para representar *streams* de entrada e de saída de bytes, isto é, cujas operações de leitura e escrita são orientadas a byte. Estas classes abstratas são estendidas de modo a fornecer diversos tipos de *streams*. *Streams* geralmente são declarados em pares, isto é, um para leitura e outro para escrita; por exemplo, a classe **FileInputStream** é utilizada para leitura de bytes de um arquivo; enquanto que a classe **FileOutputStream** permite a escrita de bytes em arquivo.

A classe abstrata **InputStream** declara métodos para leitura de bytes e é a superclasse da maioria dos *streams* de leitura orientados a byte do pacote **java.io** (ver diagrama de hierarquia de classes a seguir).




A classe **InputStream** declara entre outros métodos os seguintes:

`public abstract int read() throws IOException`

Esse método realiza a leitura de um único byte e o retorna como um inteiro no intervalo de 0 a 255. Caso não haja byte disponível para leitura devido ao fim do *stream* (por exemplo, quando o fim de arquivo for alcançado) o valor -1 será retornado.

`public int read(byte[] buff, int desloc, int cont) throws IOException`

Esse método realiza a leitura de até `cont` bytes. Os bytes são armazenados no vetor `buff` a partir da posição `desloc`. O método retorna o número de bytes efetivamente lidos ou -1 caso o fim de *stream* seja encontrado.

`public int read(byte[] buff) throws IOException`

Equivalente a `read(buff, 0, buff.length)`.

`public long skip(long cont) throws IOException`

Esse método "salta" a leitura de até `cont` bytes. O método retorna o número de bytes efetivamente "saltados".

`public int available() throws IOException`

Esse método retorna o número de bytes que podem ser lidos (ou "saltados") sem que ocorra um bloqueio. A implementação padrão retorna zero.

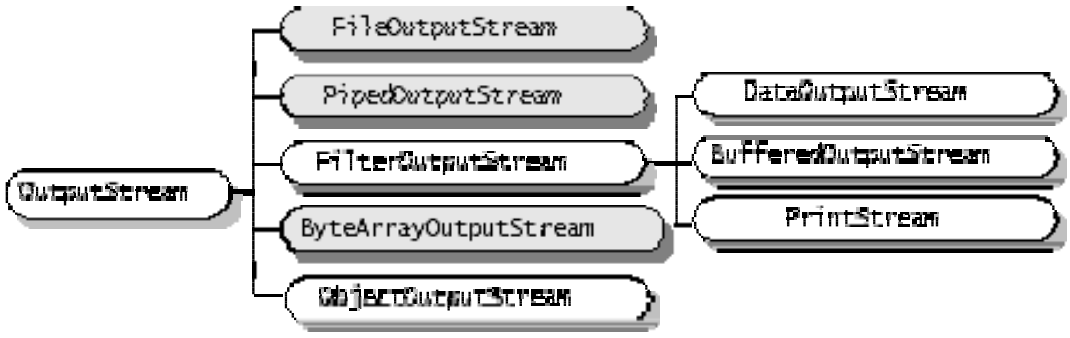
`public void close() throws IOException`

Esse método fecha o *stream*. Ele deve ser invocado de modo a liberar todos os recursos associados ao *stream* (tais como descritores de arquivos). Uma vez fechado o *stream* não deve ser utilizado e o fechamento de um *stream* já fechado não tem nenhum efeito.

O programa a seguir demonstra o uso de um *stream* de entrada para se contar o número de bytes de um arquivo :

```
import java.io.*;
class ContaBytes {
      public static void main(String[] args) throws IOException {
            FileInputStream in = new FileInputStream(args[0]);
            int total = 0;
            while (in.read() != -1)
                  total++;
            in.close();
            System.out.println(total + " bytes");
      }
}
```

A classe abstrata **OutputStream** declara métodos para escrite de bytes e é a superclasse da maioria dos *streams* de escrita orientados a byte do pacote **java.io** (ver diagrama de hierarquia de classes a seguir).




A classe **OutputStream** declara entre outros métodos os seguintes:

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

Esse método realiza a escrita de `b` como um byte, isto é, apenas os 8 bits menos significativos do inteiro `b` serão armazenados.

`public void write(byte[] buff, int desloc, int cont) throws IOException`

Esse método realiza a escrita de `cont` bytes. Os bytes devem estar armazenados no vetor `buff` a partir da posição `desloc`.

`public void write(byte[] buff) throws IOException`

Equivalente a `write(buff, 0, buff.length)`.

`public void flush() throws IOException`

Esse método "força" a escrita imediata no destino de quaisquer bytes enviados para o *stream*. Caso o destino seja outro *stream* ele também será obrigado a escrever seus bytes imediatamente (seu método `flush()` será invocado), em outras palavras, apenas uma única chamada de `flush()` é necessária para uma série de *streams* encadeados.

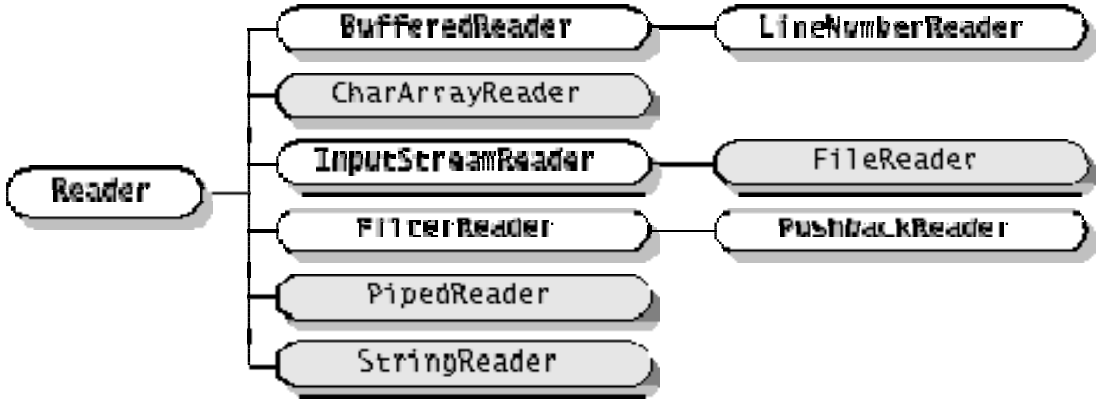
`public void close() throws IOException`

Esse método fecha o *stream*. Ele deve ser invocado de modo a liberar todos os recursos associados ao *stream* (tais como descritores de arquivos). Uma vez fechado o *stream* não deve ser utilizado e o fechamento de um *stream* já fechado não tem nenhum efeito.

## 2.3 Char Streams

O pacote **java.io** também define classes abstratas para representar *streams* de entrada e de saída de caracteres, isto é, cujas operações de leitura e escrita são orientadas a caracter. Estas classes abstratas também são estendidas de modo a fornecer diversos tipos de *streams*.

A classe abstrata **Reader** declara métodos para leitura de caracteres e é a superclasse da maioria dos *streams* de leitura orientados a caracter do pacote **java.io** (ver diagrama de hierarquia de classes a seguir).




A classe **Reader** fornece métodos similares aos declarados por **InputStream**, contudo esses método realizam a leitura de caracteres e não de bytes :

`public abstract int read() throws IOException`

Esse método realiza a leitura de um único caracter e o retorna como um inteiro no intervalo de 0 a 65535. Caso não haja caracter disponível para leitura devido ao fim do *stream* (por exemplo, quando o fim de arquivo for alcançado) o valor -1 será retornado.

`public int read(char[] buff, int desloc, int cont) throws IOException`

Esse método realiza a leitura de até `cont` caracteres. Os caracteres são armazenados no vetor `buff` a partir da posição `desloc`. O método retorna o número de caracteres efetivamente lidos ou -1 caso o fim de *stream* seja encontrado.

`public int read(char[] buff) throws IOException`

Equivalente a `read(buff, 0, buff.length)`.

`public long skip(long cont) throws IOException`

Esse método "salta" a leitura de até `cont` caracteres. O método retorna o número de caracteres efetivamente "saltados".

`public boolean ready() throws IOException`

Esse método retorna o `true` caso o *stream* esteja pronto para leitura, isto é, existe pelo um caracter para ser lido. A implementação padrão retorna zero.

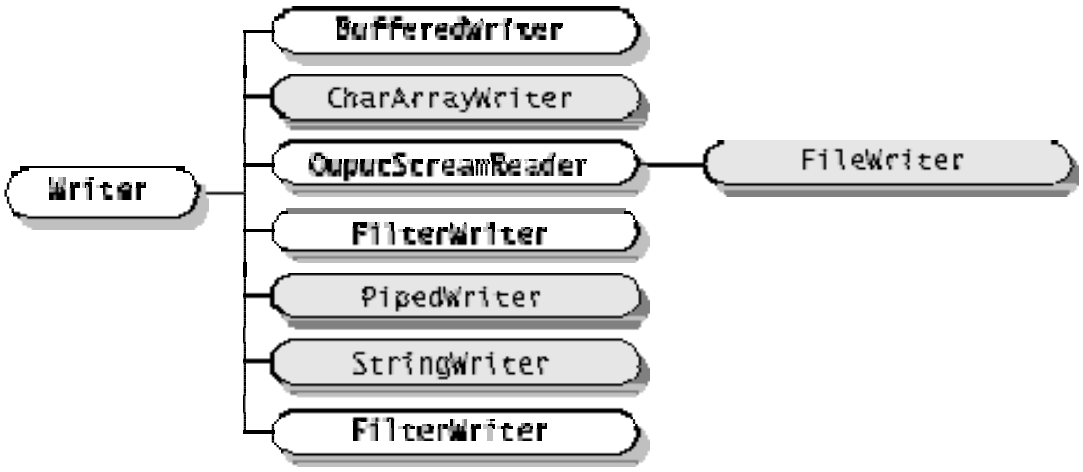
`public void close() throws IOException`

Esse método fecha o *stream*. Ele deve ser invocado de modo a liberar todos os recursos associados ao *stream* (tais como descritores de arquivos). Uma vez fechado o *stream* não deve ser utilizado e o fechamento de um *stream* já fechado não tem nenhum efeito.

O programa a seguir demonstra o uso de um *stream* de entrada para se contar o número de "whitechars" (caracteres equivalentes a um espaço ou ainda sem significado como tabulações, fim de linha, etc.) de um arquivo :

```
import java.io.*;
class ContaEspaco {
      public static void main(String[] args) throws IOException {
            FileReader in = new FileReader(args[0]);
            int total = 0, space = 0, ch;
            while ((ch = in.read()) != -1) {
                  total++;
                  if (Character.isWhitespace((char) ch))
                        space++;
            }
            in.close();
            System.out.println(total + " caracteres, " +
                              space + " espacos");
      }
}
```

A classe abstrata **Writer** declara métodos para escrita de caracteres e é a superclasse da maioria dos *streams* de escrita orientados a caracter do pacote **java.io** (ver diagrama de hierarquia de classes a seguir).



A classe **Writer** fornece métodos similares aos declarados por **OutputStream**, contudo esses método realizam a escrita de caracteres e não de bytes :

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

Esse método realiza a escrita de `ch` como um caracter, isto é, apenas os 16 bits menos significativos do inteiro `ch` serão armazenados.

`public void write(char[] buff, int desloc, int cont) throws IOException`

Esse método realiza a escrita de `cont` caracteres. Os caracteres devem estar armazenados no vetor `buff` a partir da posição `desloc`.

`public void write(char[] buff) throws IOException`

Equivalente a `write(buff, 0, buff.length)`.

`public void flush() throws IOException`

Esse método "força" a escrita imediata no destino de quaisquer caracteres enviados para o *stream*. Caso o destino seja outro *stream* ele também será obrigado a escrever seus caracteres imediatamente (seu método `flush()` será invocado), em outras palavras, apenas uma única chamada de `flush()` é necessária para uma série de *streams* encadeados.

`public void close() throws IOException`

Esse método fecha o *stream*. Ele deve ser invocado de modo a liberar todos os recursos associados ao *stream* (tais como descritores de arquivos). Uma vez fechado o *stream* não deve ser utilizado e o fechamento de um *stream* já fechado não tem nenhum efeito.

## 2.4 Standard Streams

Os *streams* que representam a entrada-padrão, a saída-padrão e o erro-padrão, respectivamente, `System.in`, `System.out` e `System.err` já existam antes da definição dos *streams* orientados a caracter, portanto foram implementados como *streams* orientados a byte apesar de logicamente serem orientados a caracteres. Esta situação causa algumas anomalias, visto que eles não poderiam (ou deveriam) ser utilizados diretamente para leitura/escrita de caracteres.

A classe **InputStreamReader** permite converter um *stream* de entrada orientado a byte em um *stream* de entrada orientado a caracter. Analogamente, a classe **OutputStreamWriter** converte um *stream* de saída orientado a byte em um *stream* de saída orientado a caracter.

O programa seguinte exemplifica o uso da classe **InputStreamReader**, juntamente com a classe **BufferedReader**, para realizar operações de leitura a partir da entrada-padrão (por exemplo, via teclado):

```
import java.io.*;
class LeTeclado {
      public static void main(String[] args) throws IOException {
            // Cria stream de conversao
            InputStreamReader fin = new InputStreamReader(System.in);

            // Cria stream "buferizado" que permite ler linhas
            BufferedReader in = new BufferedReader(fin);

            // Le uma linha a partir da entrada-padrao
            String linha = in.readLine();

            // Caso seja necessario a informacao deve ser convertida
            // por exemplo para um valor inteiro
            int valor = Integer.parseInt(linha);

            System.out.println("Valor lido : " + valor);
      }
}
```

Os objetos `System.out` e `System.err` são objetos da classe **PrintStream**. Atualmente a classe **PrintStream** foi substituída por uma versão equivalente orientada a caracter – a classe **PrintWriter**. Portanto deve-se evitar a criação de novos objetos da classe **PrintStream**.

Tanto a classe **PrintStream** como a classe **PrintWriter** fornecem métodos que tornam mais fácil a tarefa de escrever os tipos primitivos e objetos em um *stream* (em formato texto legível). Ambas as

classes fornecem os métodos **print** e **println** para os seguintes tipos: `char`, `int`, `long`, `float`, `double`, `boolean`, `char[]`, `String` e `Object`. O método **println** adiciona automaticamente um caracter de fim de linha (\n) a sua saída.

Esses métodos são muito mais convenientes que utilizar os métodos para escrita básica de caracter (uma das versões de **write**). Por exemplo, a escrita do valor de uma variável real `f` seria realizada da seguinte forma em um objeto da classe **PrintWriter** :

```
float f = 1.0;
PrintWriter out;

out.print(f);
```

Contudo, se utilizarmos o método **write** deveríamos escrever algo similar ao comando que se segue:

```
out.write(String.valueOf(f).getBytes());
```

## *2.5 Data Byte Streams*

Os *streams* orientados a byte e a caracter podem dificultar a transmissão de tipos primitivos (por exemplo, um inteiro) através de um *stream*. As interfaces **DataInput** e **DataOutput** definem métodos para transmissão (leitura e escrita, respectivamente) de tipos primitivos através de um *stream*. As classes **DataInputStream** e **DataOutputStream** fornecem uma implementação padrão para essas interfaces. Os principais métodos definidos nessas interfaces estão listados na tabela a seguir.

| Tipo | Interface | |
|---|---|---|
| | DataInput | DataOutput |
| boolean | boolean readBoolean() | void writeBoolean(boolean b) |
| char | char readChar() | void writeChar(char c) |
| byte | byte readByte() | void writeByte(byte b) |
| short | short readShort() | void writeShort(short s) |
| int | int readInt() | void writeInt(int i) |
| long | long readLong() | void writeLong(long l) |
| float | float readFloat() | void writeFloat(float f) |
| double | double readDouble() | void writeDouble(double d) |
| String | String readUTF() | void writeUTF(String s) |

As funções codificadas abaixo exemplificam o uso de tais métodos para realizar a escrita e a leitura de um vetor de reais (declarados como `double`) em um arquivo :

```
public void writeData(double[] data, String nome_arq) throws IOException
{
      // Cria stream para escrita de dados tipados em arquivo
      FileOutputStream fout = new FileOutputStream(nome_arq);
      DataOutputStream out  = new DataOutputStream(fout);
      out.writeInt(data.length);              // Escreve o tamanho do vetor
      for (int i = 0; i < data.length; i++)
            out.writeDouble(data[i]);        // Escreve cada valor
      out.close();                           // Fecha stream
}

public double[] readData(String nome_arq) throws IOException
{
      // Cria stream para leitura de dados tipados de arquivo
      FileInputStream fin = new FileInputStream(nome_arq);
      DataInputStream in  = new DataInputStream(fin);
      // Le tamanho e aloca o vetor
      double[] data = new double[in.readInt()];
      for (int i = 0; i < data.length; i++)
            data[i] = in.readDouble();          // Le cada valor
      in.close();                               // Fecha stream
      return data;
}
```

## 2.6 Object Byte Streams e Serialização de Objetos

Os *Object Streams* – **ObjectInputStream** e **ObjectOutputStream** – permitem que uma aplicação leia e escreva uma cadeia (ou "grafo") de objetos de/em um *stream*, além de permitir o armazenamento dos tipos primitivos, strings e vetores. Na realidade, ao se armazenar um objeto em um **stream**, todos os bytes representando o objeto – incluindo todos os objetos que ele referencia – são armazenados no *stream*. Este processo de transformação de um objeto em um fluxo ("*stream*") de bytes é denominado de serialização.

Para que os objetos de uma classe possam ser serializados, a classe deve implementar a interface de marcação (interface vazia) **Serializable**. No exemplo a seguir, a classe **Pessoa** implementa a interface **Serializable**, sendo portanto passível de serialização.

```
import java.io.*;

class Pessoa implements Serializable {
      private int      Id;
      private String     Nome;

      public String toString() {
         return "Id : " + Id + "\tNome : " + Nome;
      }

      public Pessoa(int Id, String Nome) {
         setId(Id);
         setNome(Nome);
      }

      public void setId(int Id) {
         this.Id = Id;
      }

      public void setNome(String Nome) {
         this.Nome = Nome;
      }

      public int getId() {
         return Id;
      }

      public String getNome() {
         return Nome;
      }
}
```

O método **writeObject** da classe **ObjectOutputStream** pode ser utilizado para serializar (escrever) um objeto de qualquer classe que implemente **Serializable**. Esse método irá armazenar todo atributo da classe no *stream*, **exceto aqueles declarados como transientes ou estáticos**.

No exemplo a seguir, um vetor de objetos da classe **Pessoa** é lido a partir do teclado e armazenado em um *stream* denominado `"person.dat"`.

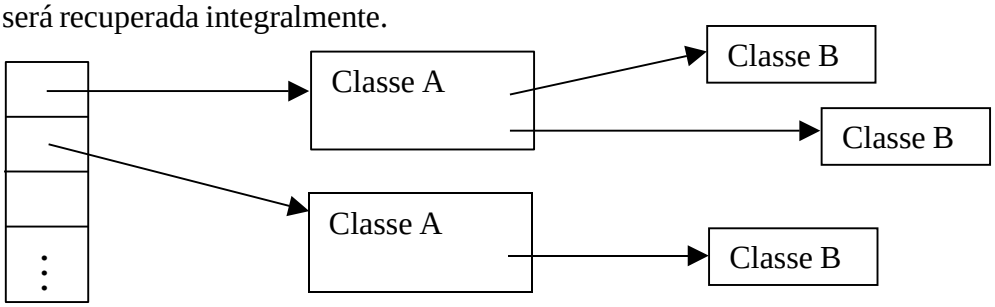
```
import java.io.*;

public class writePessoa
{
   public static void main ( String [ ] args ) throws IOException
   {
      int numElem;

      if (args.length == 0)
         numElem = 10;
      else
         numElem = Integer.parseInt(args[0]);

      // Le vetor de objetos da classe Pessoa a partir do teclado
      Pessoa[] vetor = readVetorPessoa(numElem);

      try {
            // Cria stream para escrita de objetos
           ObjectOutputStream out = new ObjectOutputStream(
                                    new FileOutputStream("person.dat"));

            // Escreve tamanho do vetor
           out.writeInt(vetor.length);
           for (int i = 0; i < vetor.length; i++)
               out.writeObject(vetor[i]);       // Escreve cada objeto

           out.close();                         // Fecha stream
      }
      catch (IOException e) {
           System.out.println("Erro de E/S !!!\n");
      }

   } // fim main ( )

   private static Pessoa[] readVetorPessoa(int numElem) throws IOException
   {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      String linha, Nome;
      int Id;

      Pessoa[] vet = new Pessoa[numElem];

      for (int i = 0; i < numElem; i++) {
         System.out.println("\nEntre com dados da " + (i+1) + "a. pessoa:");
         System.out.print("Id   : ");
         linha = in.readLine();
         Id = Integer.valueOf(linha).intValue();
         System.out.print("Nome : ");
         linha = in.readLine();
         Nome = linha;
         vet[i] = new Pessoa(Id, Nome);
      }
      return vet;
   } // fim readVetorPessoa

} // fim writePessoa class
```

O método **readObject** da classe **ObjectInputStream** pode ser utilizado para deserializar (ler) um objeto de qualquer classe que implemente **Serializable**. Esse método irá ler todo atributo da classe a partir do *stream*, **exceto aqueles declarados como transientes ou estáticos**.

No exemplo a seguir, um vetor de objetos da classe **Pessoa** é lido a partir de um *stream* denominado `"person.dat"` e exibido na saída-padrão.

```
import java.io.*;

public class readPessoa
{
   public static void main ( String [ ] args )
                              throws IOException, ClassNotFoundException
   {
      try {
            // Cria stream para leitura de objetos
           ObjectInputStream in = new ObjectInputStream(
                                    new FileInputStream("person.dat"));

            // Le tamanho e aloca vetor de objetos
           Pessoa[] vetor = new Pessoa[in.readInt()];

           for (int i = 0; i < vetor.length; i++)
               vetor[i] = (Pessoa) in.readObject(); // Le cada objeto

           in.close();                                 // Fecha stream

           System.out.println("\nLista de Pessoas :");
           System.out.println("------------------");
           for (int i = 0; i < vetor.length; i++)
             System.out.println(vetor[i]);

      }
      catch (IOException e) {
           System.out.println("Erro de E/S !!!\n");
      }

   } // fim main ( )

} // fim readPessoa class
```

A serialização preserva a integridade da cadeia (ou "grafo") de objetos. Sendo assim, caso uma aplicação possua um vetor de objetos da classe **A** cujos elementos façam referências a outros objetos, por exemplo, da classe **B** (ver diagrama abaixo); então quando o vetor for serializado (escrito em um *stream*) todos os objetos que formam a cadeia (ou "grafo") de objetos serão armazenados. Dessa forma, durante a deserialização (leitura de um *stream*) a cadeia (ou "grafo") de objetos será recuperada integralmente.



Nos exemplos seguintes considere as classes **Telefone** e **PessoaComTelefone** definidas a seguir.

```
import java.io.*;

class Telefone implements Serializable {
      private String Numero;

      public String toString() {
         return "Numero :" + Numero;
      }

      public Telefone(String Numero) {
         setNumero(Numero);
      }

      public void setNumero(String Numero) {
         this.Numero = Numero;
      }

      public String getNumero() {
         return Numero;
      }
}

class PessoaComTelefone extends Pessoa {
      private int        numTel;
      private Telefone[] Tel;

      public String toString() {
         String aux = super.toString();
         for (int i = 0; i < numTel; i++)
             aux = aux + "\n\tTel[" + (i+1) + "] " + Tel[i].toString();
         return aux;
      }

      public PessoaComTelefone(int Id, String Nome) {
         super(Id, Nome);
         numTel = 0;
      }

      public void setNumTel(int numTel) {
         this.numTel = numTel;
         Tel = new Telefone[numTel];
      }

      public int getNumTel() {
         return numTel;
      }

      public void setTel(int i, Telefone t) {
         if (i > -1 && i < numTel) Tel[i] = t;
      }

      public Telefone getTel(int i) {
         return ((i > -1 && i < numTel) ? Tel[i] : null);
      }
}
```

No exemplo a seguir, um vetor de objetos da classe **PessoaComTelefone** é lido a partir do teclado e armazenado em um *stream* denominado `"persontel.dat"`. Vale dizer que, como os objetos da classe **PessoaComTelefone** fazem referência a objetos da classe **Telefone**, estes últimos também serão armazenados no *stream* de modo que na deserialização (leitura do *stream*) a informação seja recuperada integralmente.

```
import java.io.*;
import java.util.*;

public class writePessoaTel
{
   public static void main ( String [ ] args ) throws IOException
   {
      int numElem;

      if (args.length == 0)
         numElem = 10;
      else
         numElem = Integer.parseInt(args[0]);

      // Le vetor de objetos da classe PessoaComTelefone do teclado
      PessoaComTelefone[] vetor = readVetorPessoaTel(numElem);

      try {
            // Cria stream para escrita de objetos
           ObjectOutputStream out = new ObjectOutputStream(
                                    new FileOutputStream("persontel.dat"));

            // Escreve tamanho do vetor
           out.writeInt(vetor.length);
           for (int i = 0; i < vetor.length; i++)
               out.writeObject(vetor[i]); // Escreve objeto da classe
                                          // PessoaComTelefone juntamente com
                                          // os objetos da classe Telefone

           out.close();                   // Fecha stream
      }
      catch (IOException e) {
           System.out.println("Erro de E/S !!!\n");
      }

   } // fim main ( )

   private static PessoaComTelefone[] readVetorPessoaTel(int numElem)
                                                          throws IOException
   {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      String linha, Nome;
      int Id, numTel;

      PessoaComTelefone[] vet = new PessoaComTelefone[numElem];

      for (int i = 0; i < numElem; i++) {
         System.out.println("\nEntre com dados da " + (i+1) + "a. pessoa:");
         System.out.print("Id    : ");
         linha = in.readLine();
         Id = Integer.valueOf(linha).intValue();
```

```
            System.out.print("Nome : ");
            linha = in.readLine();
            Nome = linha;
            vet[i] = new PessoaComTelefone(Id, Nome);

            System.out.print("Num.Telefones : ");
            linha = in.readLine();
            numTel = Integer.valueOf(linha).intValue();
            vet[i].setNumTel(numTel);
            for (int j = 0; j < numTel; j++) {
                System.out.print("\t" + (j+1) + "o. Numero : ");
                linha = in.readLine();
                vet[i].setTel(j, new Telefone(linha));
            }
         }
         return vet;
      } // fim readVetorPessoaTel

} // fim writePessoaTel class
```

No exemplo a seguir, um vetor de objetos da classe **PessoaComTelefone** é lido a partir de um *stream* denominado "`persontel.dat`" e exibido na saída-padrão. Todos os objetos da classe **Telefone** também serão recuperados de modo a manter a integridade das informações.

```
import java.io.*;

public class readPessoaTel
{
   public static void main ( String [ ] args )
                              throws IOException, ClassNotFoundException
   {
      try {
            // Cria stream para leitura dos objetos
           ObjectInputStream in = new ObjectInputStream(
                                    new FileInputStream("persontel.dat"));

            // Le tamanho e aloca vetor de objetos
           PessoaComTelefone[] vetor = new PessoaComTelefone[in.readInt()];

            // Le cada objeto da classe PessoaComTelefone bem como os objetos
            // da classe Telefone associados a cada um deles
           for (int i = 0; i < vetor.length; i++)
               vetor[i] = (PessoaComTelefone) in.readObject();

           in.close();                                        // Fecha stream

           System.out.println("\nLista de Pessoas :");
           System.out.println("------------------");
           for (int i = 0; i < vetor.length; i++)
             System.out.println(vetor[i]);
      }
      catch (IOException e) {
           System.out.println("Erro de E/S !!!\n");
      }

   } // fim main ( )

} // fim readPessoaTel class
```

No exemplo a seguir, o conjunto (vetor) de objetos da classe **PessoaComTelefone** será encapsulado em uma única classe de modo a criar a classe **Agenda** que será modificada mais adiante para exemplificar a customização das operações de serialização

```
import java.io.*;

class Agenda implements Serializable {
      private int                   Capacidade = 1;
      private int                   numLista;
      private PessoaComTelefone[]   ListaTel;

      public Agenda() {
         numLista = 0;
         ListaTel = new PessoaComTelefone[Capacidade];
      }

      public void inserePessoa(PessoaComTelefone p) {
           if (numLista == Capacidade) {
            PessoaComTelefone[] novaLista = new PessoaComTelefone[2*Capacidade];
            for (int i = 0; i < Capacidade; i++)
              novaLista[i] = ListaTel[i];
              Capacidade *= 2;
              ListaTel = novaLista;
         }
         ListaTel[numLista++] = p;
      }

      public int getNumPessoas() {
         return numLista;
      }

      public int getCapacidade() {
         return ListaTel.length;
      }

      public PessoaComTelefone getPessoa(int i) {
         return ((i > -1 && i < numLista) ? ListaTel[i] : null);
      }
}
```

Como todo o conjunto de objetos foi encapsulado em uma classe que implementa a interface **Serializable**, **pode-se realizar a serialização com uma única instrução**. O exemplo seguinte demonstra isto.

```
import java.io.*;

public class writeAgenda
{
   public static void main ( String [ ] args ) throws IOException
   {
      int numElem;

      if (args.length == 0)
         numElem = 10;
      else
         numElem = Integer.parseInt(args[0]);
```

```
      // Le a agenda de telefones a partir do teclado
      Agenda agendaPessoas = readAgenda(numElem);

      System.out.println("Capacidade    : " + agendaPessoas.getCapacidade());
      System.out.println("Num.Pessoas : " + agendaPessoas.getNumPessoas());

      try {
            // Cria stream para escrita dos objetos
           ObjectOutputStream out = new ObjectOutputStream(
                                    new FileOutputStream("agenda.dat"));

            // Escreve todos os objetos da agenda
           out.writeObject(agendaPessoas);

            // Fecha stream
           out.close();
      }
      catch (IOException e) {
           System.out.println("Erro de E/S !!!\n");
      }

   } // fim main ( )

   private static Agenda readAgenda(int numElem) throws IOException
   {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      String linha, Nome;
      int Id, numTel;

      Agenda ag = new Agenda();

      for (int i = 0; i < numElem; i++) {
         System.out.println("\nEntre com dados da " + (i+1) + "a. pessoa:");
         System.out.print("Id   : ");
         linha = in.readLine();
         Id = Integer.valueOf(linha).intValue();
         System.out.print("Nome : ");
         linha = in.readLine();
         Nome = linha;

         PessoaComTelefone p = new PessoaComTelefone(Id, Nome);

         System.out.print("Num.Telefones : ");
         linha = in.readLine();
         numTel = Integer.valueOf(linha).intValue();
         p.setNumTel(numTel);
         for (int j = 0; j < numTel; j++) {
             System.out.print("\t" + (j+1) + "o. Numero : ");
             linha = in.readLine();
             p.setTel(j, new Telefone(linha));
         }
         ag.inserePessoa(p);
      }
      return ag;
   } // fim readAgenda

} // fim writeAgenda class
```

Analogamente, uma única instrução pode ser utilizada para deserializar os todos os objetos da agenda. O exemplo seguinte demonstra isto.

```
import java.io.*;

public class readAgenda
{
   public static void main ( String [ ] args )
                              throws IOException, ClassNotFoundException
   {
      try {
            // Cria stream para leitura dos objetos
           ObjectInputStream in = new ObjectInputStream(
                                    new FileInputStream("agenda.dat"));

            // Le o objeto da classe Agenda e todos os demais objetos a ele
            // associados (classes PessoaComTelefone e Telefone)
           Agenda agendaPessoas = (Agenda) in.readObject();

            // Fecha stream
           in.close();

           System.out.println("Capacidade    : " + agendaPessoas.getCapacidade());
           System.out.println("Num.Pessoas : " + agendaPessoas.getNumPessoas());

           System.out.println("\nLista de Pessoas :");
           System.out.println("------------------");
           for (int i = 0; i < vetor.length; i++)
             System.out.println(vetor[i]);*/
      }
      catch (IOException e) {
           System.out.println("Erro de E/S !!!\n");
      }

   } // fim main ( )

} // fim readAgenda class
```

### 2.6.1 Customização da Serialização de Objetos

Nos dois exemplos anteriores envolvendo a classe **Agenda**, a capacidade do vetor de objetos da classe **PessoaComTelefone** vai sendo dobrada sempre que necessário. Sendo assim após ter armazenado 10 pessoas a capacidade será igual a 16. Logo ficará evidente o grande desperdício por detrás desta implementação. Uma forma de contornarmos esse problema, seria não armazenar a capacidade do vetor durante a serialização. Isto pode ser feito declarando-se o atributo **Capacidade** como transiente, uma vez que qualquer atributo declarado como estático ou transiente não será armazenado durante a serialização. Contudo para que a classe **Agenda** funcione corretamente após sua leitura (deserialização) devemos fornecer um valor para o atributo **Capacidade** (já que no *stream* não haverá informação sobre ele).

Para tanto devemos fornecer versões "customizadas" dos métodos **writeObject** e **readObject** para a classe **Agenda**. Esse métodos devem ser declarados como privados e "sem retorno" (void). O método **writeObject** recebe um **ObjectOutputStream** como parâmetro e deve declarar a possível

ocorrência de exceções da classe **IOException**. Já o método **readObject** recebe um **ObjectInputStream** como parâmetro e deve declarar a possível ocorrência de exceções da classe **IOException** e da classe **ClassNotFoundException**. No exemplo a seguir, a classe **Agenda** foi modificada de modo a se declarar o atributo **Capacidade** como transiente e fornecer implementações próprias ("customizadas") dos métodos **writeObject** e **readObject**.

```
import java.io.*;
class AgendaModified implements Serializable {
      transient private int         Capacidade = 1;
      private int                   numLista;
      private PessoaComTelefone[]   ListaTel;

      public AgendaModified() {
         numLista = 0;
         ListaTel = new PessoaComTelefone[Capacidade];
      }

      public void inserePessoa(PessoaComTelefone p) {
           if (numLista == Capacidade) {
            PessoaComTelefone[] novaLista = new PessoaComTelefone[2*Capacidade];
            for (int i = 0; i < Capacidade; i++)
              novaLista[i] = ListaTel[i];
              Capacidade *= 2;
              ListaTel = novaLista;
         }
         ListaTel[numLista++] = p;
      }

      public int getNumPessoas() {
         return numLista;
      }

      public int getCapacidade() {
         return ListaTel.length;
      }

      public PessoaComTelefone getPessoa(int i) {
         return ((i > -1 && i < numTel) ? ListaTel[i] : null);
      }

      private void writeObject(ObjectOutputStream out) throws IOException {
         out.writeInt(numLista);
         for (int i = 0; i < numLista; i++)
            out.writeObject(ListaTel[i]);
      }

      private void readObject(ObjectInputStream in)
           throws IOException, ClassNotFoundException
      {
         numLista = in.readInt();
         Capacidade = numLista;
         ListaTel = new PessoaComTelefone[Capacidade];
         for (int i = 0; i < numLista; i++)
            ListaTel[i] = (PessoaComTelefone) in.readObject();
      }
}
```

Vale observar que nas versões "customizadas" dos métodos **readObject** e **writeObject** da classe **AgendaModified**, utilizaram-se os métodos das interfaces **ObjectInput** e **ObjectOutput** para realizar a leitura e escrita dos atributos da classe. Essas interfaces estendem as interfaces **DataInput** e **DataOutput** permitindo não só a leitura e escrita de tipos primitivos mas também de objetos.

O exemplo seguinte demonstra a serialização "customizada" em funcionamento.

```
import java.io.*;

public class writeAgendaMod
{
   public static void main ( String [ ] args ) throws IOException
   {
      int numElem;

      if (args.length == 0)
         numElem = 10;
      else
         numElem = Integer.parseInt(args[0]);

      // Le a agenda de telefones a partir do teclado
      AgendaModified agendaPessoas = readAgenda(numElem);

      System.out.println("Capacidade    : " + agendaPessoas.getCapacidade());
      System.out.println("Num.Pessoas : " + agendaPessoas.getNumPessoas());

      try {
            // Cria stream para escrita dos objetos
           ObjectOutputStream out = new ObjectOutputStream(
                                    new FileOutputStream("agenda.dat"));

            // Escreve todos os objetos da agenda
           out.writeObject(agendaPessoas);

            // Fecha stream
           out.close();
      }
      catch (IOException e) {
           System.out.println("Erro de E/S !!!\n");
      }

   } // fim main ( )

   private static AgendaModified readAgendaMod(int numElem) throws IOException
   {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      String linha, Nome;
      int Id, numTel;

      AgendaModified ag = new AgendaModified();

      for (int i = 0; i < numElem; i++) {
         System.out.println("\nEntre com dados da " + (i+1) + "a. pessoa:");
         System.out.print("Id   : ");
         linha = in.readLine();
         Id = Integer.valueOf(linha).intValue();
         System.out.print("Nome : ");
```

```
            linha = in.readLine();
            Nome = linha;

            PessoaComTelefone p = new PessoaComTelefone(Id, Nome);

            System.out.print("Num.Telefones : ");
            linha = in.readLine();
            numTel = Integer.valueOf(linha).intValue();
            p.setNumTel(numTel);
            for (int j = 0; j < numTel; j++) {
                System.out.print("\t" + (j+1) + "o. Numero : ");
                linha = in.readLine();
                p.setTel(j, new Telefone(linha));
            }
            ag.inserePessoa(p);
         }
         return ag;
      } // fim readAgendaMod

} // fim writeAgendaMod class
```

Analogamente, uma única instrução pode ser utilizada para deserializar os todos os objetos da agenda. O exemplo seguinte demonstra isto.

```
import java.io.*;
public class readAgendaMod
{
   public static void main ( String [ ] args )
                              throws IOException, ClassNotFoundException
   {
      try {
            // Cria stream para leitura dos objetos
           ObjectInputStream in = new ObjectInputStream(
                                    new FileInputStream("agenda.dat"));

            // Le o objeto da classe AgendaModifief e todos os demais objetos
            // a ele associados (classes PessoaComTelefone e Telefone)
           AgendaModified agendaPessoas = (AgendaModified) in.readObject();

            // Fecha stream
           in.close();

           System.out.println("Capacidade    : " + agendaPessoas.getCapacidade());
           System.out.println("Num.Pessoas : " + agendaPessoas.getNumPessoas());

           System.out.println("\nLista de Pessoas :");
           System.out.println("------------------");
           for (int i = 0; i < vetor.length; i++)
             System.out.println(vetor[i]);*/
      }
      catch (IOException e) {
           System.out.println("Erro de E/S !!!\n");
      }

   } // fim main ( )

} // fim readAgendaMod class
```

A princípio, as duas últimas implementações apresentadas podem parecer idênticas as implementações anteriores para escrita e leitura da agenda, contudo uma análise mais cuidadosa logo exibirá suas diferenças.

Como mencionado anteriormente, nos dois exemplos anteriores envolvendo a classe **Agenda**, a capacidade do vetor de objetos da classe **PessoaComTelefone** vai sendo dobrada sempre que necessário. Sendo assim após ter armazenado 10 pessoas a capacidade será igual a 16. Logo ficará evidente o grande desperdício por detrás daquela implementação.

A nova implementação utilizando a classe **AgendaModified** também dobra a capacidade sempre que for necessário. Contudo esse atributo não é armazenado durante a serialização. Ao se realizar a leitura da agenda, a quantidade de pessoas efetivamente armazenada é utilizada como capacidade inicial (isto é, o vetor de objetos da classe **PessoaComTelefone** terá capacidade para armazenar exatamente o número de objetos que foram gravados no *stream*). Porém caso seja necessário aumentar a capacidade do vetor, tudo funcionará como antes (ou melhor, o vetor terá sua capacidade dobrada).

Utilizando os números apresentados anteriormente, após ter armazenado 10 pessoas a capacidade do vetor será igual a 16, entretanto ao se serializar a classe **AgendaModified** a capacidade não será armazenada. Durante a leitura (deserialização) da classe **AgendaModified** a capacidade será inicializada com o número atual de elementos do vetor, isto é, 10. Porém, se alguma inserção for realizada a capacidade será dobra, ou melhor, se tornará igual a 20.

### 2.6.2 Controle de Versão

As implementações de classe mudam ao longo do tempo. Caso haja mudanças na implementação de uma classe entre o momento em que um objeto foi serializado e o momento em que ele é deserializado, a classe **ObjectInputStream** é capaz detectar essas mudanças. Quando um objeto é escrito, um identificador único denominado "serial version UID (unique identifier)" é armazenado junto com ele.

Esse identificador é um inteiro longo (64 bits) que por padrão é obtido através de uma função hash segura envolvendo o nome completo da classe, suas superinterfaces, seus atributos e métodos, de modo que se algum desses elementos for alterado uma possível incompatibilidade entre as classes será sinalizada. Esse identificador atua como uma "impressão digital" fazendo com que seja praticamente impossível que duas classes distintas tenha o mesmo valor de UID.

Quando um objeto é lido a partir de um ObjectInputStream, sua identificador também é lido. Em seguida, é feita uma tentativa de se carregar a classe (correspondente ao objeto). Caso não se encontre nenhuma classe com o mesmo nome ou caso o UID da classe carregado não seja igual ao UID armazenado no stream, o método readObject "arremessa" uma exceção da classe InvalidClassException. Caso se encontre versões de todas as classes utilizadas na declaração do objeto e se todos os UIDs forem iguais, o objeto poderá ser deserializado.

Essa abordagem é extremamente conservadora, pois qualquer mudança na declaração da classe pode criar uma versão totalmente incompatível com anterior. Entretanto muitas das alterações em uma classe não são tão extremas a ponto de torná-la totalmente incompatível. Quando se fizer uma alteração em uma classe que continua compatível com as formas serializadas de versões anteriores

da classe, pode-se declarar explicitamente para essa classe o seu "serial version UID". Tal declaração deve ser feita da seguinte forma :

```
private static final long serialVersionUID = <valor>;
```

O valor do identificador é fornecido pelo sistema de desenvolvimento. Na maioria dos ambientes, ele pode ser obtido através do aplicativo `serialver`. Nada impede o uso de qualquer número como identificador desde que você o declare a partir da primeira versão da classe, contudo geralmente é uma péssima idéia, pois seu número não será calculado de forma cuidadosa para evitar conflito com outras classes. Portanto o ideal é utilizar o valor de UID fornecido pelo ambiente de desenvolvimento antes da alteração.

**OBSERVAÇÃO:** A compatibilidade efetiva das operações de serialização e deserialização fica a cargo do programador, isto é, eventualmente será necessário fornecer versões "customizadas" dos métodos **readObject** e **writeObject** para que a leitura e escrita funcionem corretamente.