Tratamento de Exceções Entrada e Saída

Tecnicas de Programacao

José Antonio F. de Macêdo
jose.macedo@lia.ufc.br

MANIPULAÇÃO DE EXCEÇÕES

- Sua função é:
- transferir controle de onde ocorreu o erro para um manipulador de erro que possa lidar com a situação

- OBS:
  - Semelhante a C++ e Delphi
  - Mais flexível que On Error GoTo de VB

Exemplos de Exceções

- Erros de entrada do usuário
- digitação
- Erros de dispositivo
- impressora sem papel
- Limitações físicas
- disco cheio
- Erros de código
- acesso a pilha vazia

EXCEÇÕES

- Sinalizam condições de erro
- *Exceções Explícitas*
- devem ser tratadas
- sinalizam condições contornáveis
- *Exceções Implícitas*
- não precisam ser tratadas
- sinalizam condições geralmente incontornáveis

EXCEÇÕES

- Exceções são representadas por objetos
- Exceções são subclasses de **Throwable**
- **Exception**
- tipo base para *checked exceptions*
- **Error** & **RuntimeException**
- tipo base para *unchecked exceptions*
- **throw** lança uma exceção

EXCEÇÕES EXPLÍCITAS

- Precisam ser declaradas pelos métodos

```
class ListaE {
  private ListaE próximo;
  public void insere(ListaE e) throws Exception {
    if (e == null)
      throw new Exception("Elemento nulo");
    e.próximo = próximo;
    próximo = e;
  }
}
```

EXCEÇÕES IMPLÍCITAS

- Não precisam ser declaradas pelos métodos

```
class ListaE {
  private ListaE próximo;
  public void insere(ListaE e) {
    if (e == null)
      throw new Error("Elemento nulo");
    e.próximo = próximo;
    próximo = e;
  }
}
```

EXCEÇÕES EXPLÍCITAS

- Precisam ser tratadas pelos métodos

```
class ListaE {
  public void insere(ListaE e) throws Exception {
    ...
  }
  public void duploInsere(ListaE e1, ListaE e2) {
    insere(e2); // ERRO!
    insere(e1); // ERRO!
  }
}
```

EXCEÇÕES IMPLÍCITAS

- Não precisam ser tratadas pelos métodos

```
class ListaE {
  public void insere(ListaE e) {
    ... throw Error("Elemento nulo"); ...
  }
  public void duploInsere(ListaE e1, ListaE e2) {
    insere(e2); // OK!
    insere(e1); // OK!
  }
}
```

O QUE PRECISAMOS SABER ...

- Capturar as exceções lançadas pelos métodos que tentamos executar.
  - Comandos: **try, catch, finally**
- Anunciar uma exceção a qual pode ser lançada por um método.
  - Comando: **throws**
- Lançar e repassar exceções dentro de métodos.
  - Comando: **throw**

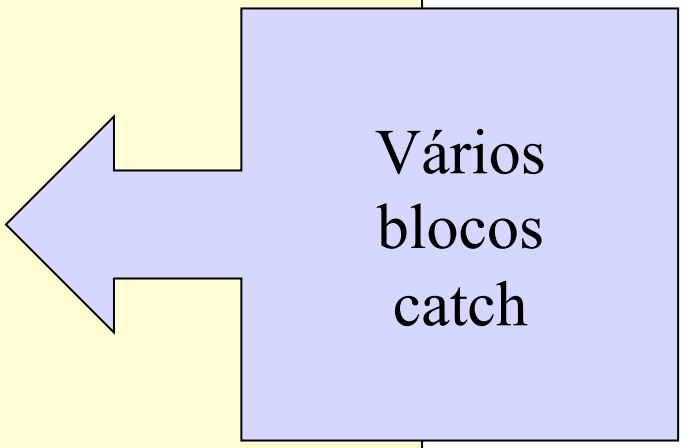
CAPTURAR EXCEÇÕES LANÇADAS

- Tentar executar um bloco de código
- No caso de erro, capturar exceções que foram lançadas
- finalmente realizar algum tipo de limpeza

```
try {
    // … execucar algo que possa causar uma exceção
}
catch ( TipoExcecao variavel) {
    // … tratar exceção
}
finally {
    // … ao final executar sempre este código
}
```

É possível tratar diversas exceções !

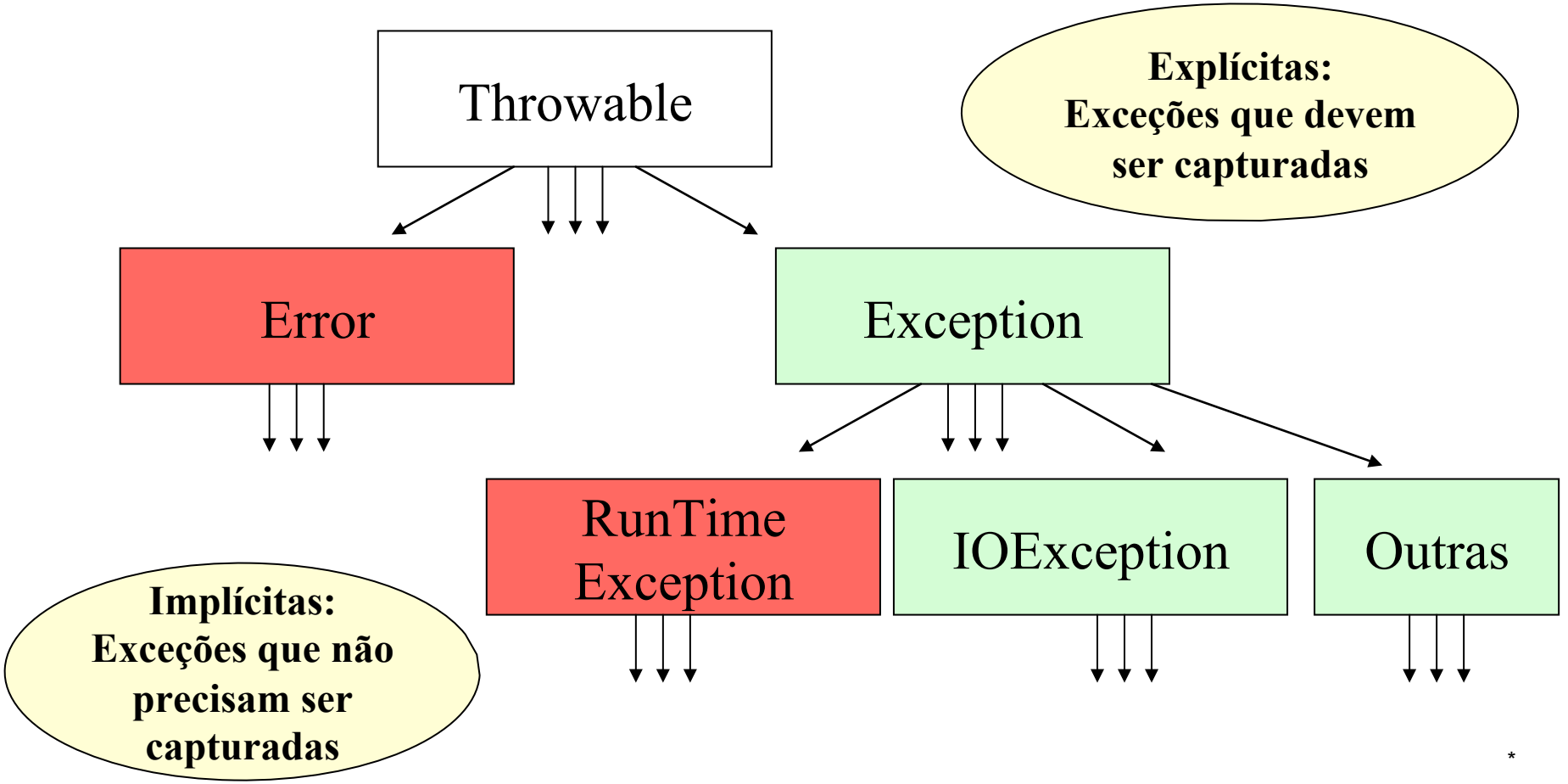
```
try {
   // … execucar algo que possa causar uma exceção
}
catch ( TipoExcecao1 variavel1) {
   // … tratar exceção1
}
catch ( TipoExcecao2 variavel2) {
   // … tratar exceção2
}
catch ( TipoExcecao3 variavel3) {
   // … tratar exceção3
}
finally {
  // … ao final executar sempre este código
}
```



Exemplo - Captura de exceções

```
int divisao (int a, int b) {
    int resultado = 0;
    try {
      resultado = a / b;
    }
    catch (ArithmeticException e) {
      System.out.println ("\tOcorreu um erro de divisão");
    }
    finally {
      System.out.println ("\tSempre vai apresentar esta mensagem");
    }
    return resultado;
  }
```

Hierarquia de Herança de Exceção

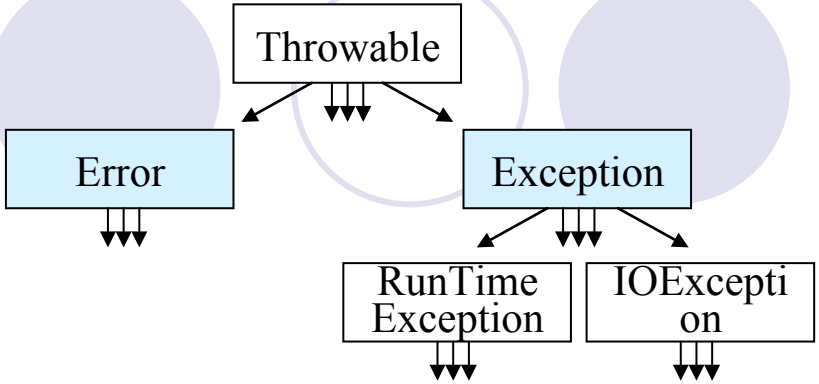


IMPORTANTE

- Exceção Implícita:
- derivada de Error ou de RunTimeException
- Exceção Explícita:
- todas as outras

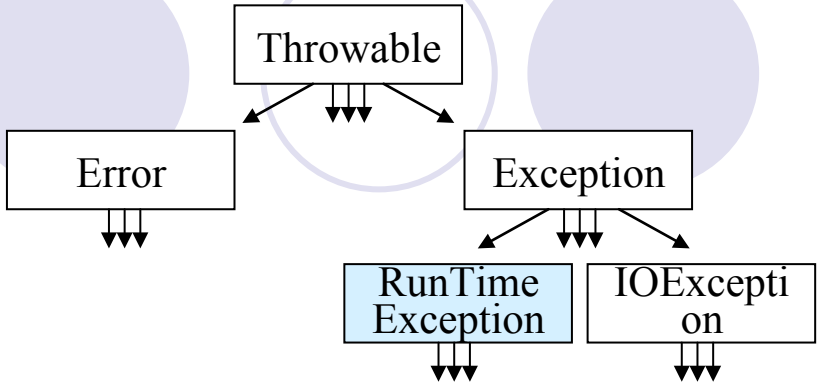
- Importante:
- Implícitas = fora de seu controle ou derivam de condições que deveriam ter sido tratadas por vc
- Explícitas = são as que vc deve tratar!
- Um método deve declarar todas as exceções explícitas que passa

COMENTÁRIOS



- Throwable
- Error
  - descreve erros internos e falta de recursos em tempo de execução (raro)
  - programador não trata nem passa objetos deste tipo
- Exception
  - Programador deve tratar esta hierarquia
  - RuntimeException:
    - conversor errado, acesso a array fora dos limites
    - regra: A CULPA é NORMALMENTE do programador !
  - IOException
    - tentativa de ler além do fim de arquivo, URL mal formatada
  - *Suas classes de exceção*

RUNTIMEEXCEPTION



- RuntimeException
- NullPointer, Security, NegativeArrayIndex, Arithmetic, IndexOutOfBounds, ...
- *não é necessário tentar capturar estas exceções*
- *não é necessário especificá-las na cláusula throws*
- Outras subclasses de Exception
- ou
  - lida com exception onde ela pode ocorrer (try { ... } catch { ... })
- ou
  - passa adiante, mas deve fazer declaração em throws

ANUNCIANDO UMA EXCEÇÃO

Método não apenas informa valores a serem retornados, informa também o que pode sair errado

Indica que este método pode lançar uma exceção

```
public static void sleep (long t) throws InterruptedException
```

Tipo da exceção a ser lançada

Um método que executar o método ***sleep*** deve:

- 1. capturar a exceção lançada pelo ***sleep*** usando ***try-catch***

ou

- 2. anunciar a exceção lançada pelo ***sleep*** e repassá-la usando o comando ***throw***.

CAPTURANDO UMA EXCEÇÃO

- capturar a exceção lançada pelo ***sleep*** usando ***try-catch***

```
…
void meuMetodo ( ) {
    try {
            x.sleep(10);
    }
    catch (InterruptedException e) {
            // Tratamento
    }
}
…
```

EXEMPLO DE TRY-CATCH-FINALLY

```
void readFile(String name) throws IOException {
  FileReader file = null;
  try {
    file = new FileReader(name);
    ... // lê o arquivo
  } catch (Exception e) {
    System.out.println(e);
  } finally {
    if (file != null) file.close();
  }
}
```

REPASSANDO UMA EXCEÇÃO

- anunciar a exceção lançada pelo ***sleep*** e repasso-a.

```
…
void meuMetodo ( ) throws InterruptedException {
    x.sleep(10);
}
…
```

Quem executar o método ***meuMetodo( )*** deverá capturar e tratar a exceção ***InterruptedException***

REPASSANDO UMA EXCEÇÃO (OUTRA FORMA)

- anunciar a exceção lançada pelo ***sleep*** e repasso usando o comando ***throw***.

```
…
void meuMetodo ( ) throws InterruptedException {
    try {
            x.sleep(10);
    }
    catch (InterruptedException e) {
            // Faço alguma coisa ...
            throw e;
    }
}
…
```

Repasso a exceção capturada

Quem executar o método ***meuMetodo( )*** deverá capturar e tratar a exceção ***InterruptedException***

Quais exceções um método pode lançar ?

- Um método só pode lançar as exceções explícitas listadas na cláusula ***throws*** ou subclasses dessas exceções
- Um método pode lançar qualquer exceção implícita, mesmo que ela não tenha sido declarada na cláusula ***th***

```
…
void meuMetodo ( ) throws Exception {
    if ( i == 0 )
       throw new IOException();
    if (i == -1)
       throw new ArithmeticException ();
}
…
```

CRIANDO NOVAS EXCEÇÕES

- Para criar novas exceções basta criar classes que extendam as classes de exceções ou suas subclasses.
- Dependendo da superclasse voce poderá ter uma exceção implícita ou explícita.

```
class MinhaExcecao extends Exception {

}
```

USANDO AS NOVAS EXCEÇÕES

```
class testeExcecao {
   void meuMetodo ( ) throws MinhaExcecao {
   }

   void outroMetodo ( ) {
         try {
               meuMetodo();
         } catch (MinhaExcecao e) {
               // trato
         }
   }
}
```

```
class MinhaExcecao
      extends Exception {

}
```

Dica de Depuração

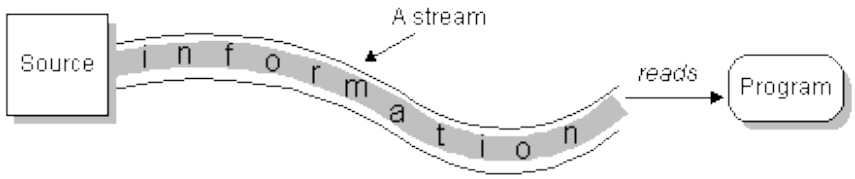
- Objetos de Exceção
- podem imprimir “stack trace” para permitir verificar exceção
- fornecem msg (string) com alguns detalhes
- podem ter outras propriedades específicas de subclasses

```
try
{
    …
}
catch (Throwable t)
{
    t.printStackTrace ();
    throw t;
}
```

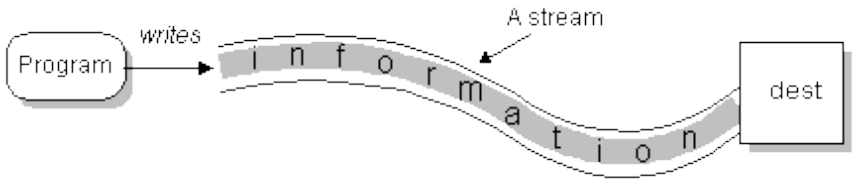
Entrada e Saída

Pacote Padrão *java.io*

- Abstração do sistema nativo
- *Streams* de entrada e saída
- *Streams* de *bytes* e de caracteres
- **InputStream/Reader**

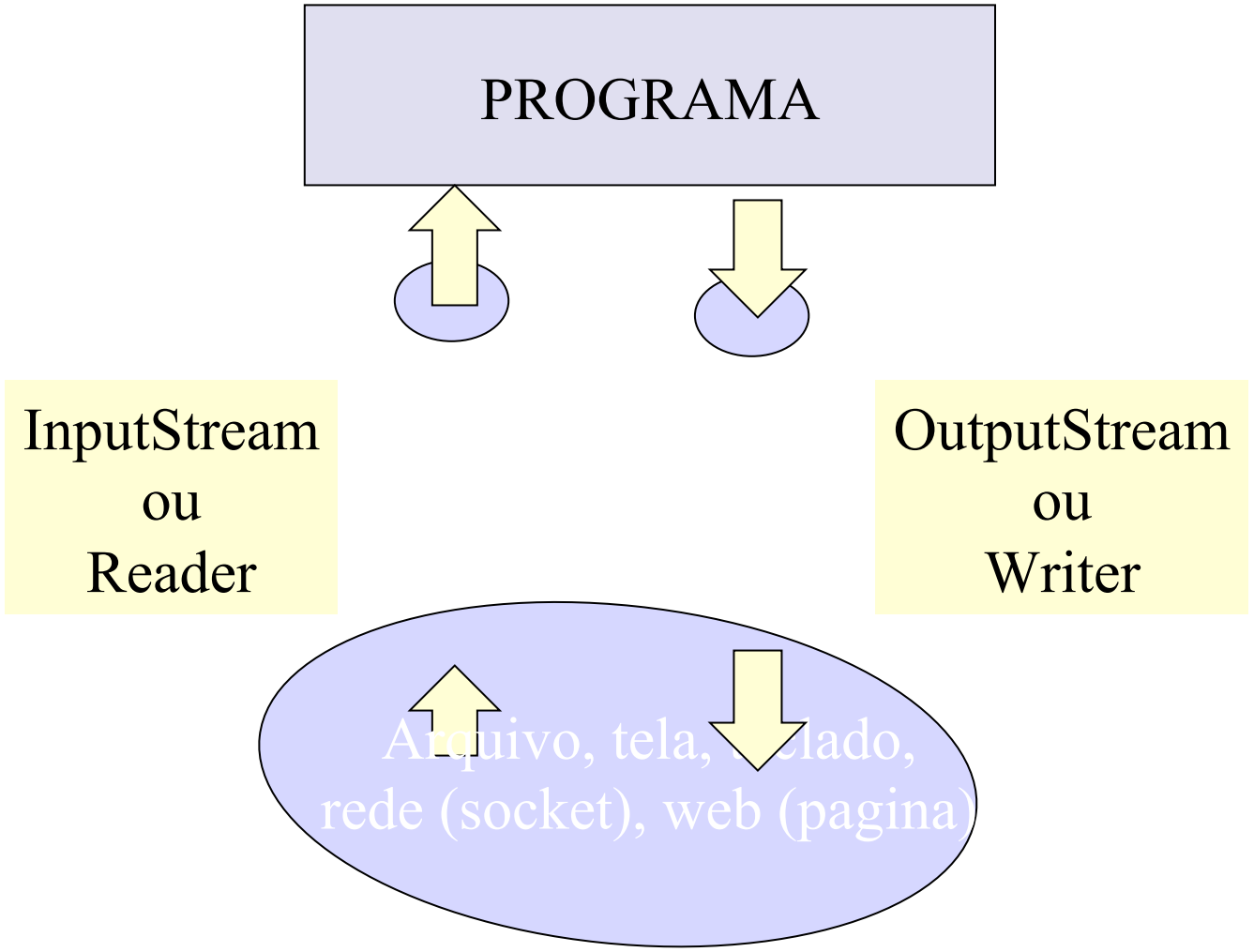


- **OutputStream/Writer**



- ***Streams* de dados**
- ***Streams* de objetos**

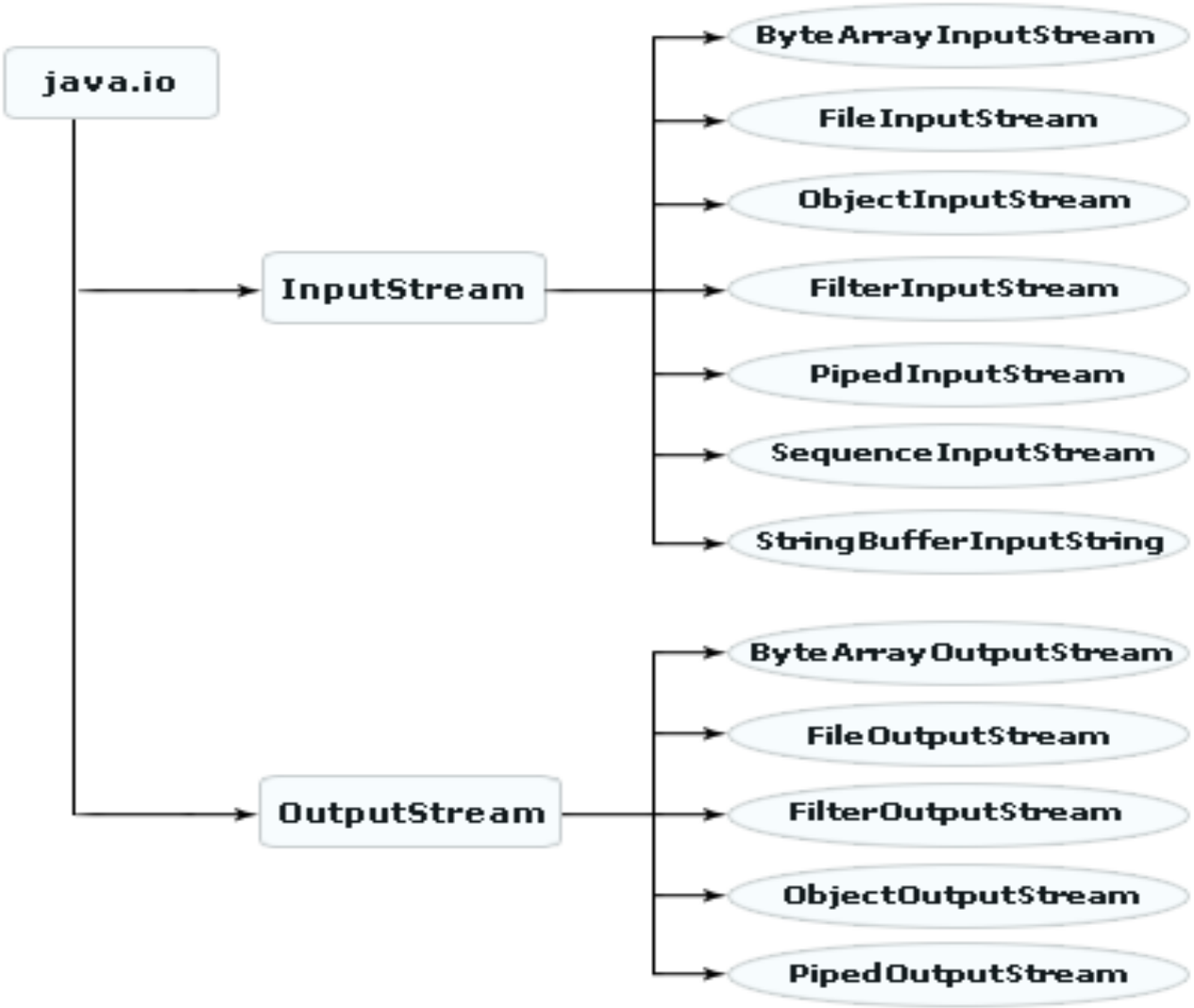
FLUXOS DE ENTRADA E SAÍDA



INTRODUÇÃO

- Input e Output fazem parte do pacote java.io
- Esse pacote tem uma quantidade grande de classes que dão suporte a operações de entrada e saída
- As classes básicas são InputStream e OutputStream

Hierarquia de classes do java.io



*

Tipos de Streams

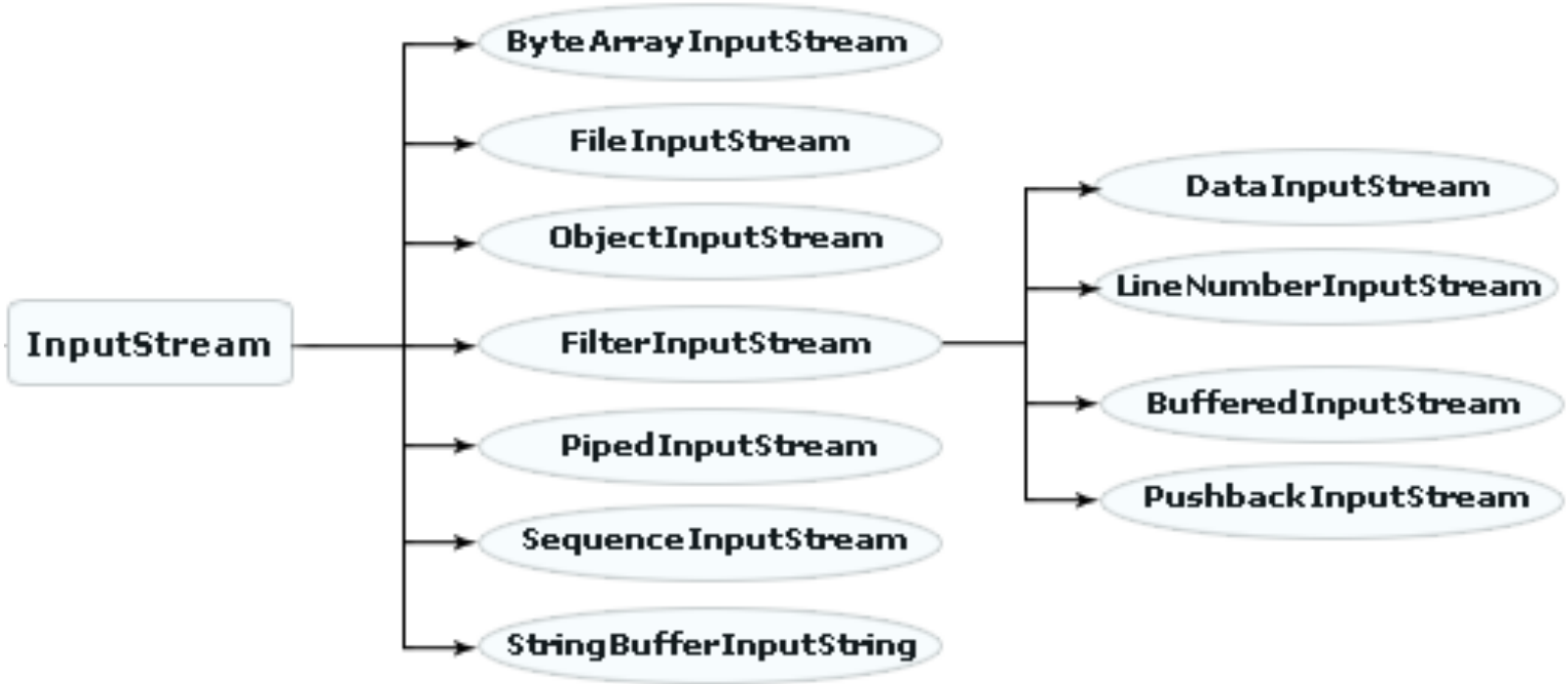
- Existem dois tipos de Streams: binário e texto
- No primeiro a gente pode transferir bytes (arquivos, por exemplo), no segundo, caracteres

INPUTSTREAM

- InputStream é usada para ler dados como bytes de uma fonte de dados (Teclado, Arquivo, Rede, etc)
- É uma classe abstrata
- Voce deve fechar seu stream apos o uso, ou esperar que seja coletado pelo Garbage Collector

HIERARQUIA DE INPUTSTREAM

- Você deve usar uma dessas subclasses
- Cada uma tem um propósito diferente



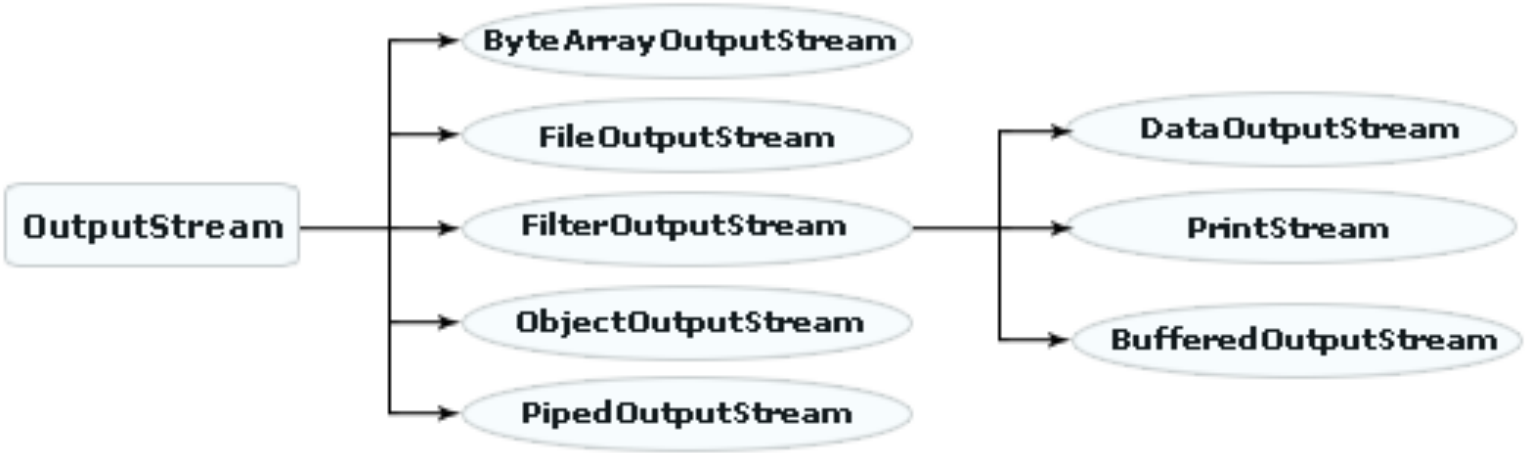
*

OUTPUTSTREAM

- OutputStream é similar ao InputStream
- É usada para escrever bytes em alguma fonte
- Você pode ler do disco e gravar na memória, ler um objeto e gravar no disco...
- Todas as combinações são válidas, você faz de acordo com a conveniência

HIERARQUIA

- Mais uma vez você só pode usar uma das subclasses de OutputStream



PRINCIPAIS CLASSES

- Classe **File**
- Acesso via *streams*
- **FileInputStream**
- **FileOutputStream**
- **FileReader**
- **FileWriter**
- Acesso aleatório
- **RandomAccessFile**

FILE

- Essa classe trabalha com os arquivos do computador de forma independente
- Dessa forma, você pode escrever código que manipula arquivos independentemente da plataforma
- Um objeto do tipo File pode representar tanto um arquivo como um diretório
- Quando ele é criado, o SO não checa a existência efetiva do arquivo/diretório

JAVA.IO.FILE

```
public File(String path)
public String getName()
public String getPath()
public boolean exists()
public boolean canWrite()
public boolean canRead()
public boolean isFile()
public boolean isDirectory()
public boolean renameTo(File dest)
public boolean delete()
```

PRINCIPAIS MÉTODOS

- f.exists() – true se o arquivo existir
- f.isFile() – true se é um arquivo
- f.isDirectory() – true se é um diretório
- f.getName() – nome do arquivo/diretório
- f.length() – número de bytes de um arquivo
- f.getPath() – nome do caminho
- f.delete() – apaga
- f.renameTo(f2) – renomeia para f2
- f.createNewFile() – cria o arquivo (IOException)

EXEMPLO DO USO DE FILE

```
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;

public class TrabalhandoComArquivos {
  public static void main(T[] args) throws Exception {
    File arquivo = new File("c:/install.log");
    System.out.println(“Existe: ” + arquivo.exists());
    System.out.println(“Nome do arquivo: ” + arquivo.getName());
    System.out.println(“Tamanho em bytes: “ + arquivo.length());
  }
}
```

Lendo arquivos dentro de um diretorio

```
import java.io.*;
public class LeitorDeArquivos {
  public static void main(String[] args) {
    File diretorio = new File(”/MeuDir/Java");
    if (diretorio.isDirectory()) {
      for (String nomeDoArquivo : diretorio.list()) {
        String caminho = diretorio.getPath();
        File arquivo = new File(caminho + "/" + nomeDoArquivo);
        if (arquivo.isFile()) {
          System.out.print(arquivo.getName() + " - ");
          long tamanhoEmMB = arquivo.length() / 1024;
          System.out.println(tamanhoEmMB + "MB");
        }
      }
    }}}
```

JAVA.IO.INPUTSTREAM

```
public InputStream()
public abstract int read() throws IOException
public int read(byte[] buf) throws IOException
public long skip(long n) throws IOException
public int available() throws IOException
public void close() throws IOException
public synchronized void mark(int readlimit)
public synchronized void reset() throws IOException
public boolean markSupported()
```

INPUTSTREAM

- **abstract int read() throws IOException**
- lê um byte de dados . retorna -1 ao final do fluxo

- **void close() throws IOException**
- fecha o fluxo de saída. Importante: porque um fluxo consome muitos recursos do S.O.

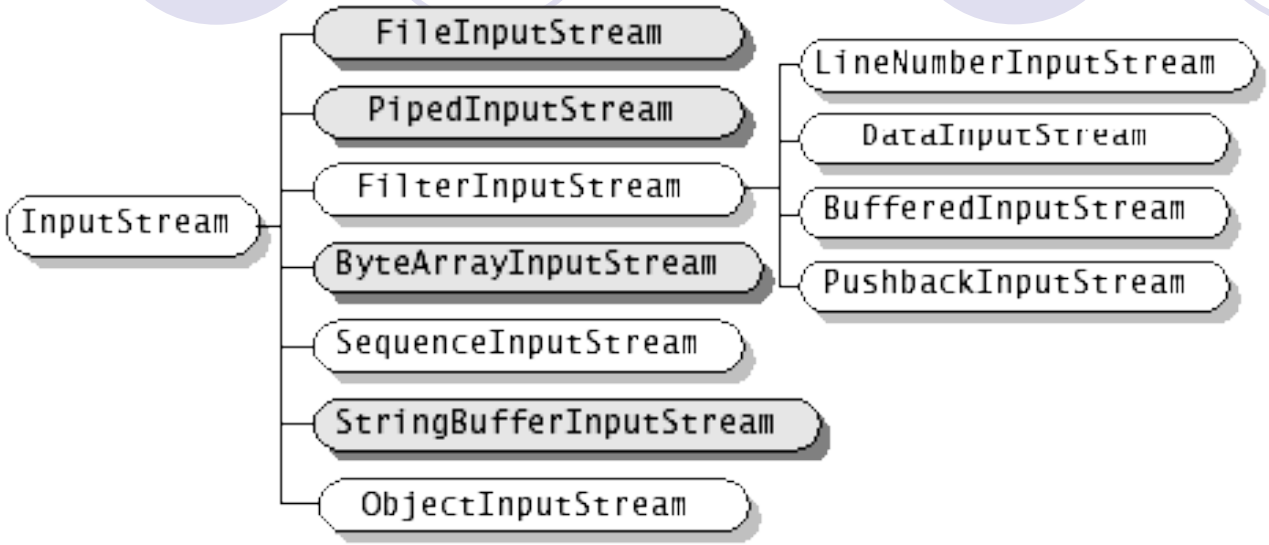
JAVA.IO.OUTPUTSTREAM

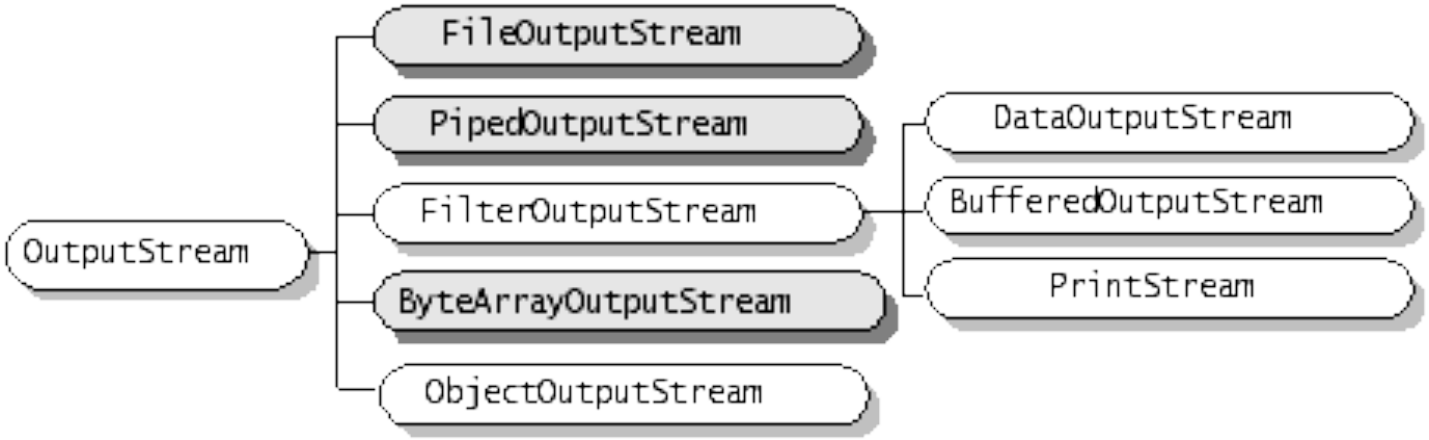
```
public OutputStream()
public abstract void write(int b) throws IOException
public void write(byte[] buf) throws IOException
public void flush() throws IOException
public void close() throws IOException
```

OUTPUTSTREAM

- **abstract void write(int b) throws IOException**
- envia um byte
- **void close() throws IOException**
- fecha o fluxo de saída
- **void flush() throws IOException**
- esvazia o fluxo de saída, envia todos os dados armazenados no buffer

Hierarquia: InputStream, OutputStream





*

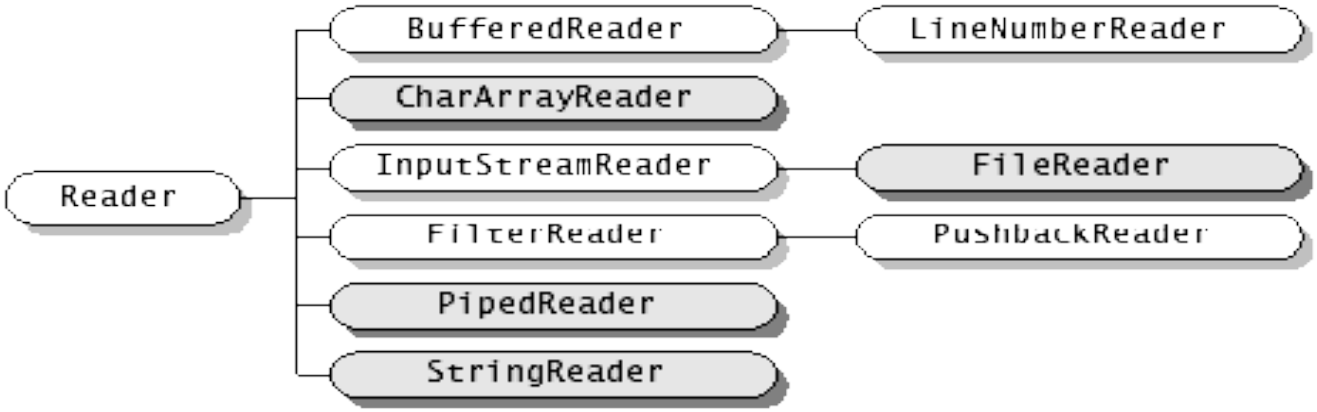
JAVA.IO.READER

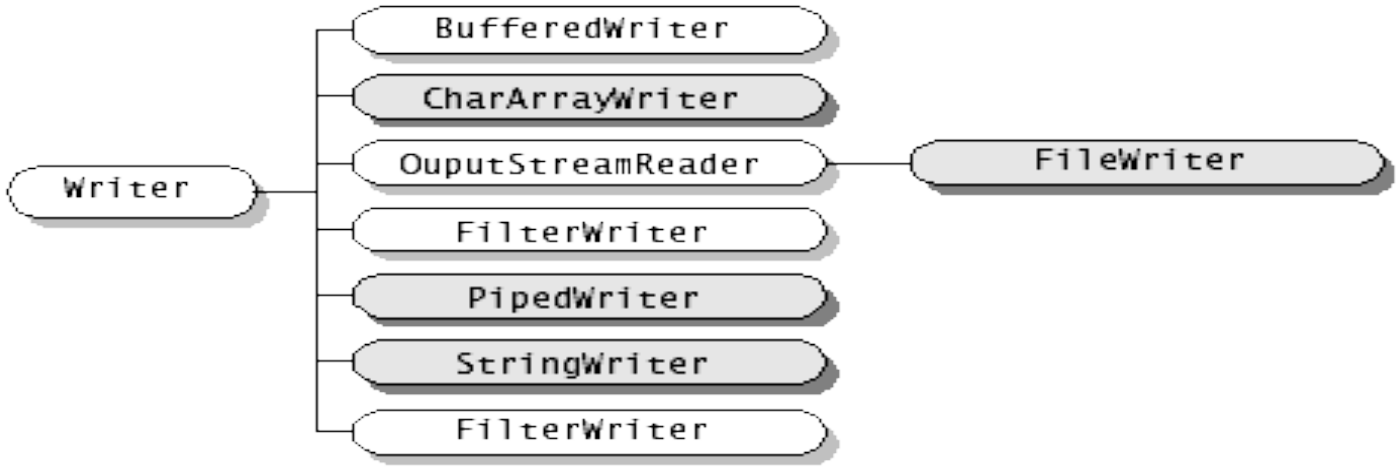
```
public Reader()
public Reader(Object lock)
public int read() throws IOException
public int read(char[] buf) throws IOException
public long skip(long n) throws IOException
public boolean ready() throws IOException
public abstract void close() throws IOException
public void mark(int readlimit)
public void reset() throws IOException
public boolean markSupported()
```

JAVA.IO.WRITER

```
public Writer()
public Writer(Object lock)
public void write(int c) throws IOException
public void write(char[] buf) throws IOException
public void write(String str) throws IOException
public abstract void flush() throws IOException
public abstract void close() throws IOException
```

Hierarquia: Reader, Writer





ESCREVENDO ARQUIVOS

- Uma operação bastante utilizada no desenvolvimento de software é a criação de arquivos
- Podemos fazer de diversas formas diferentes
- Para tanto, vamos começar com arquivos texto mais simples
- As classes usadas: FileOutputStream e FileInputStream

EXEMPLO CÓPIA DE ARQUIVO USANDO FILEINPUTSTREAM E FILEOUTPUTSTREAM

```
File inputFile = new File(“origem.txt");
File outputFile = new File(“destino.txt");

FileInputStream in = new FileInputStream(inputFile);
FileOutputStream out = new FileOutputStream(outputFile);
int c;

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

in.close();
out.close();
```

EXEMPLO CÓPIA DE ARQUIVO USANDO FILEREADER E FILEWRITER

```
File inputFile = new File(“origem.txt");
File outputFile = new File(“destino.txt");

FileReader in = new FileReader(inputFile);
FileWriter out = new FileWriter(outputFile);
int c;

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

in.close();
out.close();
```

Escrevendo arquivos

```
import java.io.File;
import java.io.FileOutputStream;

public class EscrevendoArquivos {
  public static void main(String[] args) {
    try {
      File f = new File("c:/NovoArquivo.txt");
      FileOutputStream fo = new FileOutputStream(f);
      String texto = “Vamos gravar este Texto no arquivo";
      texto = texto + "\nEsta é a segunda linha";
      fo.write(texto.getBytes());
      fo.close();
      System.out.println("Arquivo gravado com sucesso");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

LENDO ARQUIVOS

- Para ler um arquivo (ou qualquer outra fonte de dados) através da stream, devemos recuperar uma linha de cada vez, ou seja, fazer um loop até chegar no fim dos dados
- Existem diversas variações para realizar uma leitura de arquivo

EXEMPLO DE LEITURA DE ARQUIVO

```
try {
  Reader r = new FileReader(“teste.txt”);
  int c;
  while ((c = r.read()) != -1) {
    System.out.println(“Li caracter ”+c);
  }
}
catch (FileNotFoundException e) {
  System.out.println(“teste.txt não existe”);
}
catch (IOException e) {
  System.out.println(“Erro de leitura”);
}
```

Lendo arquivos

```
import java.io.File;
import java.io.FileInputStream;

public class LendoArquivos {
  public static void main(String[] args) {
    try {
                        File f = new File("c:/NovoArquivo.txt");
                        FileInputStream fi = new FileInputStream(f);
            int i = 0;
                        while(i!=-1){
        i = fi.read();
                          char c = (char) i;
        System.out.print(c);
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

LEITURA DO TECLADO

```
import java.io.*;
public class ReadStIS
{
   public static void main(String[] args) throws Exception
   {  char c;
      System.out.println("Informe um caracter:");
      c = (char)System.in.read();
      System.out.println("Primeiro caracter: " + c);
      int bytesProntos = System.in.available();
      System.out.println("resto dos bytes: " + bytesProntos);
      if (bytesProntos > 0) {
        byte[] entrada = new byte[bytesProntos];
        System.in.read(entrada);
        for (int i=0; i <= entrada.length-1; i++)
          System.out.println("Voce informou: " + (char)entrada[i]);
      } }}
```

Resultado da Execucao de ReadStIS

Informe um caracter:
Jose Antonio
Primeiro caracter: J
resto dos bytes: 12
Voce informou: o
Voce informou: s
Voce informou: e
Voce informou:
Voce informou: A
Voce informou: n
Voce informou: t
Voce informou: o
Voce informou: n
Voce informou: i
Voce informou: o
Voce informou:

*

STREAMS DE DADOS

- Definidos por interfaces
- **DataInput**
- **DataOutput**
- Permitem escrita e leitura de tipos básicos (char, float, integer, double, etc)
- Implementados por
- **DataInputStream**
- **DataOutputStream**
- **RandomAccessFile**

JAVA.IO.DATAINPUT

```
public abstract void readFully(byte b[]) throws
IOException
public abstract int skipBytes(int n) throws
IOException
public abstract boolean readBoolean() throws
IOException
public abstract byte readByte() throws IOException
public abstract int readUnsignedByte() throws
IOException
public abstract char readChar() throws IOException
...
```

JAVA.IO.DATAOUTPUT

```
public abstract void write(byte b[]) throws
IOException
public abstract void writeBoolean(boolean v) throws
IOException
public abstract void writeByte(int v) throws
IOException
public abstract void writeChar(int v) throws
IOException
public abstract void writeInt(int v) throws
IOException
...
```

EXEMPLO: CRIANDO ARQUIVO

```
char lineSep = System.getProperty("line.separator").charAt(0);

// Cria arquivo ped.txt
DataOutputStream out = new DataOutputStream(new
FileOutputStream(“ped.txt"));
double[] prices = { 19.99, 9.99, 15.99};
int[] units = { 12, 8, 13 };
String[] descs = { “Camisa", “Sapato",“Chaveiro” };

for (int i = 0; i < prices.length; i ++) {
      out.writeDouble(prices[i]);   out.writeChar('\t');
      out.writeInt(units[i]);            out.writeChar('\t');
      out.writeChars(descs[i]);     out.writeChar(lineSep);
 }
out.close();
```

EXEMPLO: LENDO ARQUIVO

```
 // Le o arquivo novamente
DataInputStream in = new DataInputStream(new FileInputStream(“ped.txt"));

double price;   int unit;  StringBuffer desc; double total = 0.0;  char chr;

try {
    while (true) {
        price = in.readDouble();       in.readChar();       // pula tabulação
        unit = in.readInt();               in.readChar();       // pula tabulação
                while ((chr = in.readChar()) != lineSep)
                 desc.append(chr);
        System.out.println(“Ped. " +unit + " unid. de " + desc + "  a R$" + price);
        total = total + unit * price;
    }
}
catch (EOFException e) { }

System.out.println(“Total de: R$" + total);
in.close();
```

EXEMPLO: EXECUÇÃO DO PROGRAMA

Ped. 12 unid. de Camisa a R$19.99
Ped. 8 unid. de Sapato a R$9.99
Ped. 13 unid. de Chaveiro a R$15.99
For a TOTAL of: $527.67

Fluxos de Impressão: Gravando arquivo Texto (PrintStream)

```
//Abrir um arquivo para gravação
FileOutputStream arqsai = new fileOutputStream("alunos.dat");

//Abrindo um fluxo de impressão
PrintStream saidaTexto = new PrintStream(arqsai);
saidaTexto.println("Joao, 23, 1000");
saidaTexto.println("Maria,30,450");
arqsai.close();
```

ESCRITA DE ARQUIVO TEXTO FORMATADO (JAVA 5)

```
import java.io.*; java.util.*;
public class PrintWriterFileApp{
  public static void main (String arg[]) {
      File file = new File ("textOutput.txt");
                  Formatter formatter;
      try {
        formatter = new Formatter (file);
        formatter.format ("Saida com PrintWriter. %n");
        formatter.format ("tipos primitivos convertidos em strings: %n");
        boolean a_boolean =  false;
        int  an_int =  1234567;
                  double a_double = -4.297e-15;
       formatter.format ("boolean = %9b %n", a_boolean);
       formatter.format ("int     = %9d %n", an_int);
                  formatter.format ("double  = %9.2e %n", a_double);
                  formatter.flush ();
    formatter.close ();
      }
      catch (IOException ioe){  System.out.println("IO Exception"); }
  }}
```

STREAMS

- Podem atuar sobre vetores
- **ByteArrayInputStream**
- **ByteArrayOutputStream**
- **CharArrayReader**
- **CharArrayWriter**

```
public ByteArrayInputStream(byte[] buf)
public ByteArrayOutputStream(int buf_size)
```

STREAMS

- Podem atuar sobre *strings*
- **StringReader**
- **StringWriter**

```
public StringReader(String str)
public StringWriter(int buf_size)
```

STREAMS

- Podem ser “buferizados”
- **BufferedInputStream**
- **BufferedOutputStream**
- **BufferedReader**
- **BufferedWriter**

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int size)
```

Lendo do buffer

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;

public class LendoLinhasInteiras {
  public static void main(String[] args) {
    try {
                        FileInputStream fi = new FileInputStream(”NovoArquivo.txt");
      InputStreamReader ir = new InputStreamReader(fi);
              BufferedReader br = new BufferedReader(ir);
      String linha;
                        while ((linha = br.readLine()) != null) {
        System.out.println(linha);
       }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

LEITURA DO TECLADO BUFFERIZADA

```
public static void echofile() {
    String buffer = " ";
    InputStreamReader teclado = new InputStreamReader(System.in);
    BufferedReader bufferin = new BufferedReader(teclado);
    while (true)
    {
       try{
            buffer = bufferin.readLine();
       }
       catch(IOException e)
       {
            System.out.println("Erro durante a leitura" + e);
            return;
       }
      if( buffer == null ) break;
      System.out.println(buffer); // Mostra na tela
    }
}
```

SERIALIZANDO OBJETOS

- Em algumas situações precisamos transportar objetos pela rede. Antes disso, o objeto precisa ser gravado no disco (serializado), depois é só enviar por um socket para o computador remoto
- Vamos criar uma classe de teste e serializá-la no disco
- Depois a gente recupera a mesma, preservando seu estado

Classe de teste

```
import java.io.Serializable;
public class AlgumaClasse implements Serializable {
  private String nome;
  private String outroNome;
  private String maisOutroNome;

  public String getNome() { return nome;}
  public void setNome(String nome) { this.nome = nome;}
  public String getOutroNome() { return outroNome; }
  public void setOutroNome(String outroNome) {
    this.outroNome = outroNome;
  }
  public String getMaisOutroNome() {
    return maisOutroNome;
  }
  public void setMaisOutroNome(String maisOutroNome) {
    this.maisOutroNome = maisOutroNome;
  }
}
```

Serializando

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
public class SerializandoObjetos {
  public static void main(String[] args) {
    try {
      AlgumaClasse a = new AlgumaClasse();
         a.setMaisOutroNome("a");
         a.setNome("b");
         a.setOutroNome("c");
      FileOutputStream fo = new FileOutputStream(”classe.tmp");
      ObjectOutputStream ou = new ObjectOutputStream(fo);
      ou.writeObject(a);
      ou.close();
      fo.close();
      System.out.println("Objeto serializado");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Deserializar

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class DeserializandoObjetos {
  public static void main(String[] args) {
    try {
      FileInputStream fi = new FileInputStream(”classe.tmp");
      ObjectInputStream oi = new ObjectInputStream(fi);
      Object o = oi.readObject();
      AlgumaClasse a = (AlgumaClasse) o;
      System.out.println(a.getMaisOutroNome());
            System.out.println(a.getNome());
      System.out.println(a.getOutroNome());
      oi.close();
      fi.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

PrintWriter

- Classe para impressão rápida de caracteres em um OutputStream, inclusive fazendo formatação
- No nosso exemplo, a saída vai ser um arquivo texto
- Existe outra classe chamada PrintStream que trabalha com bytes

Fluxo de Impressão PrintStream

- Imprime strings e números em formato texto
- Métodos:

void print(String s)
void println(String s)
void print(char s[])
void print(char c)
void print (boolean b)
void print(int i)
void print(long l)
void print(float f)
void print(double d)

PrintWriter

```
import java.io.File;
import java.io.PrintWriter;

public class UsandoPrintWriter {
  public static void main(String[] args) {
    try {
      File f = new File(”ArquivoPrintWriter.txt");
      PrintWriter p = new PrintWriter(f);
      p.print("primeira linha");
      p.println("segunda linha");
      p.write("terceira linha");
      p.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Acesso Randomico

- Permite realizar leituras e gravacoes de maneira randomica
- Classe principal RandomAccessFile

ACESSO RANDOMICO

```
import java.io.*;
public class TesteRandom{
  public static void main(String argv[]){
    try {
      TesteRandom r = new TesteRandom();
      RandomAccessFile raf = new RandomAccessFile(“random.txt","rw");

      r.escreve(raf);
      r.leUm(raf,2);
      r.escreveUm(raf,2,'x');
      r.leUm(raf,2);

      raf.close();
    } catch(IOException ioe) {System.out.println(ioe);}
  }
```

ACESSO RANDOMICO (CONT.)

```
public void escreve(RandomAccessFile raf) throws IOException {
    char[] letras = {'a', 'b', 'c', 'd'};
    for(int i=0; i<letras.length;i++){
      raf.writeChar(letras[i]);
    }
  }
  public void leUm(RandomAccessFile raf, int pos) throws IOException {
    raf.seek(pos);
    System.out.println(raf.readChar());
  }
  public void escreveUm(RandomAccessFile raf, int pos, char c) throws IOException {
    raf.seek(pos);
    raf.writeChar(c);
  }
}
```

CODIFICADORES DE STRING

- Usados para manipular strings, procurando delimitadores e separando em pedaços:
- Implementado pela classe StringTokenizer
- Construtor:
- StringTokenizer(String str, String delim)
  - str: string de entrada.
  - Delim: delimitador dos tokens

STRINGTOKENIZER

- Boolean hasMoreTokens()
- String nextToken()
- retorna o próximo token

- int countTokens()
- retorna o número de tokens existentes na string

EXEMPLO

```
FileInputStream arqsai = new FileInputStream(“alunos.dat”);
DataInputStream arq = DataInputStream(arqsai);

String reg = arq.readLine();
StringTokenizer registro = new StringTokenizer(reg,”,”);
while (registro.hasMoreTokens())
{
        String campo = registro.nextToken();
}
```