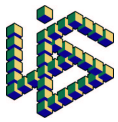


I/O em Java

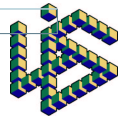
- É baseado na idéia de fluxos de dados (*streams).*

Java I/O

Fábio R. de Miranda

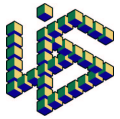
LIVES
Laboratório de Imagens, Visão e Estruturas Espaciais
Faculdade SENAC de Ciências Exatas e Tecnologia

Setembro de 2003



Streams de informação

- Programa acessando um *stream* de entrada



I/O em Java

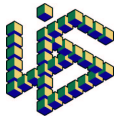
- O Java tem um modelo poderoso e flexível de I/O
- Por ser tão poderoso, às vezes ele não parece muito simples

Streams de saída

- De maneira análoga, uma aplicação pode enviar informação para um destino externo abrindo um *stream* para este destino e escrevendo a informação seqüencialmente

Streams de entrada

- Para trazer informação até ele, um programa acessa um fluxo de dados de alguma fonte de informação (arquivos, memória, sockets, etc) e a lê seqüencialmente, como podemos ver no próximo slide.

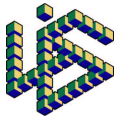


Streams

Semelhança de procedimentos de leitura e escrita

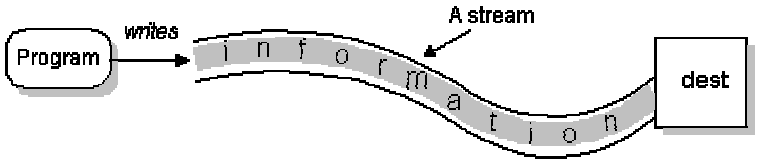
Entrada:
abrir um stream
while (mais informações)
  ler
fechar o *stream*

Saída:
abrir um stream
while (mais informações)
  escrever
fechar o *stream*



Streams de informação

- Programa enviando dados para um *stream*

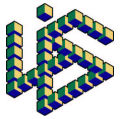


Fluxos de dados

- Os dois tipos de *streams* que poderemos ter são os orientados a caracteres (lêem e escrevem de arquivos texto) e os orientados a bytes (lêem e escrevem de 8 em 8 bits)

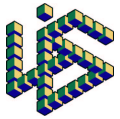
O pacote java.io

- Este pacote tem diversas classes de leitura e escrita para *streams*. Para usá-las, um programa precisa importar o pacote java.io.



Caracteres: Reader e Writer

- int read()
- int read(char cbuf[])
- int read(char cbuf[], int offset, int length)
- int write(int c)
- int write(char cbuf[])
- int write(char cbuf[], int offset, int length)



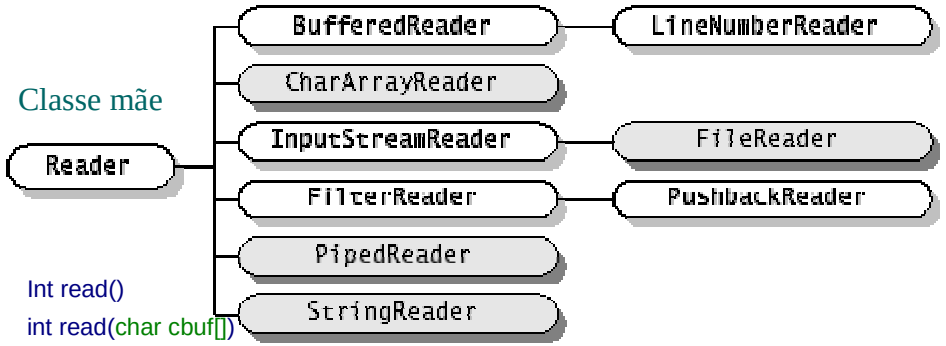
Duas hierarquias em java.io

- O pacote java.io divide-se em duas hierarquias de classes, dependendo do tipo de dados que constituem os *streams* em que operam.

Bytes (8 bits): InputStream e OutputStream

- int read()
- int read(byte bbuf[])
- Int read(byte bbuf[], int offset, int length)
- int write(int c)
- int write(byte bbuf[])
- int write(byte bbuf[], int offset, int length)

Readers: entrada baseada em caracteres

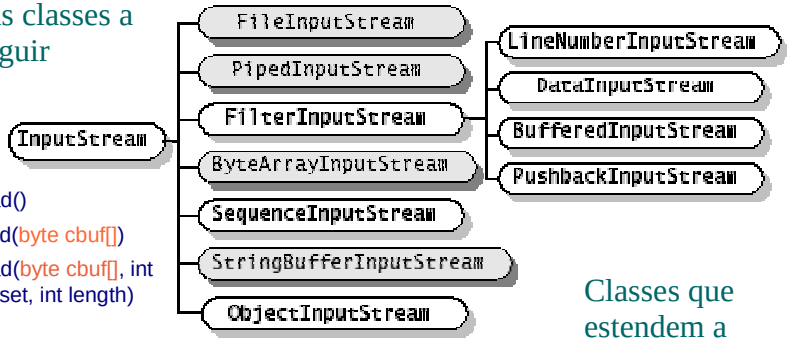


Int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)

Classes filhas: Customizam e acrescentam funcionalidade

InputStreams: entrada baseada em bytes

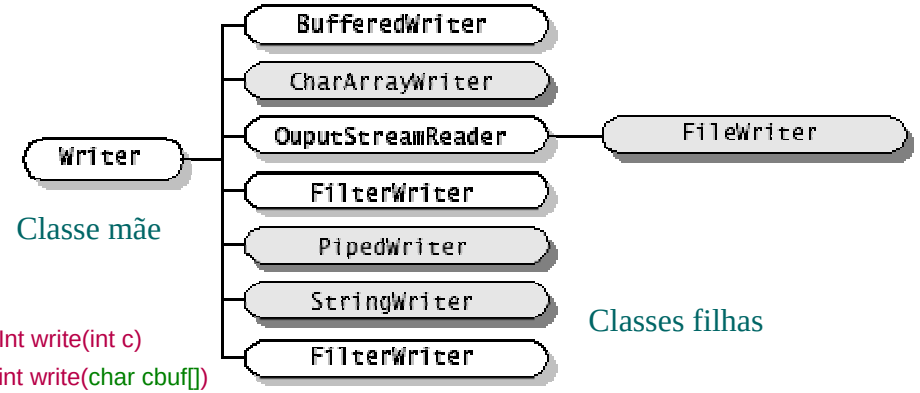
Superclasse das classes a seguir



Int read()
int read(byte cbuf[])
Int read(byte cbuf[], int offset, int length)

Classes que estendem a funcionalidade

Writers: Saída baseada em caracteres



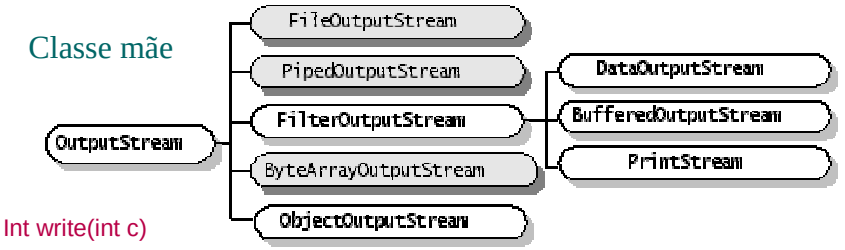
Int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)

Classes filhas

Exemplo de File Streams

```
import java.io.*;

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

      FileInputStream in = new FileInputStream("entrada.txt");
      FileOutputStream out = new FileOutputStream("saida.txt");
      int c;

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

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

OutputStreams: saída baseada em bytes



Int write(int c)

int write(byte cbuf[])

int write(byte cbuf[], int offset, int length)

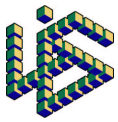
A classe File

- A classe java.io.File permite que criemos objetos que representam um caminho (*path*) até um arquivo. Usá-lo pode ser muito útil para desenvolver software multiplataforma, em que os *paths* têm representações diferente.
- O programa do próximo slide utiliza o objeto File para realizar a mesma tarefa do programa do slide anterior.

Streams de e para arquivos

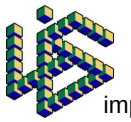
Os fluxos de dados de e para arquivos estão entre os de uso mais comum e fácil:

- FileInputStream
- FileOutputStream
- FileReader
- FileWriter



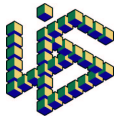
“Embrulhando” um stream

- Em alguns momentos a API de I/O permite que passemos *streams* como argumentos para outros *streams*. Ao fazermos isso estamos “embrulhando” o primeiro com o segundo.
- Em geral fazemos isso para ganhar alguma funcionalidade.



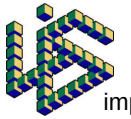
Exemplo de File Streams

```
import java.io.*;

public class CopyBytes {
   public static void main(String[] args) throws IOException {
      File inputFile = new File("entrada.txt");
      File outputFile = new File("saida.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();
   }
}
```



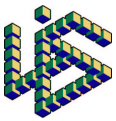
Embrulhos típicos

- BufferedReader – Otimiza os acessos a um *InputStream* fazendo várias leituras ao mesmo tempo. Serve para otimizar os acessos a um InputStream fazendo várias leituras ao mesmo tempo.
- FilterInputStream – Acessa um *InputStream* com a opção de fazer alguma transformação nos dados lidos, um exemplo é o *GZipInputStream*



Exemplo de File Streams (caracteres)

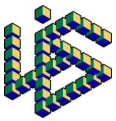
```
import java.io.*;

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


      FileReader in = new FileReader(“entrada.txt”);
      FileWriter out = new FileWriter(“saida.txt”);
      int c;

      while ((c = in.read()) != -1)
        // read() retornará -1 quando o stream acabar
        out.write(c);

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



Exemplos

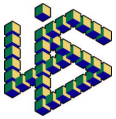
ConsoleIO.java:

```
BufferedReader inputReader = new
  BufferedReader(new
  InputStreamReader(System.in));
...
 while (!leitura.equals("fora")){
    leitura = inputReader.readLine();
   out.println("O que você escreveu foi:
  "+leitura);
 }
```



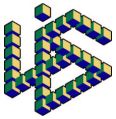
Embrulhos típicos

- InputStreamReader – Serve para acoplar a entrada do mundo voltado a bytes para o voltado a caracteres
- OutputStreamWriter – Acopla a saída do mundo voltado a caracteres ao voltado a bytes



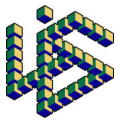
Leitura a partir do console

- No exemplo anterior, embrulhamos System.in - um objeto de uma subclasse de InputStream – primeiro em um InputStreamReader, para converter sua entrada em caracteres, depois em um BufferedReader, para podermos ler a entrada em caracteres linha a linha



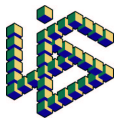
Embrulhando o console

- A maneira recomendada de se fazer a leitura de dados do console em Java é “embrulhá-lo” em um BufferedReader.
- Para fazer a escrita, recomenda-se o embrulho em um PrintWriter, apesar de PrintStream (o tipo do System.out) poder ser usado normalmente na maior parte das vezes).



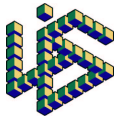
Streams de impressão

- Os *streams* de impressão formatam a saída para leitura humana. Esta é uma tarefa claramente voltada a caracteres (é o que lemos, afinal). Por motivos históricos, *System.out* ainda é um *stream* voltado a bytes (*PrintStream*).



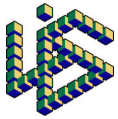
Leitura a partir do console

- Um BufferedReader, através de chamadas ao método readLine(), retorna sempre uma String que contém a linha digitada pelo usuário.
- Para convertê-la em um dado de interesse, deveremos utilizar um método de *parseXXX(String)* conveniente de uma das *wrapper classes*: Double, Float, Integer, Long.



Separando a Entrada

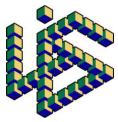
- StringTokenizer – Permite definir caracteres separadores e quebrar uma String em diversas substrings
- StreamTokenizer – Permite trabalhar num *stream* de entrada, dividindo-o em *tokens*.



Exemplo de leitura e conversão

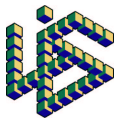
- LeituraNumerica.java

```
linha = reader.readLine();
idade = Integer.parseInt(linha);
```



Stream Tokenizer

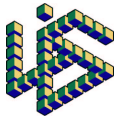
- No exemplo anterior, o método nextToken() retorna um inteiro indicando se o que foi encontrado no stream de entrada foi um inteiro, uma palavra, uma quebra de linha ou um fim de arquivo.
- O método *toString()*, ao ser chamado em uma instância da classe *StreamTokenizer*, retorna uma identificação do último token.



StringTokenizer

- TestaStringTokenizer.java

```
String str = "Keep walking.Johnnie
Walker. Keep.";
StringTokenizer tokenizer = new
StringTokenizer(str, " ,");
      while(tokenizer.hasMoreTokens()){

System.out.println(tokenizer.nextToken(
));
}
```



Gravando e lendo objetos

- O ambiente Java permite que objetos sejam enviados através de um *stream* e posteriormente restaurados.
- Para dotar os objetos que criarmos de capacidade de escrita em disco, devemos implementar a interface *Serializable*

StreamTokenizer

TestaStreamTokenizer.java

```
        while (token!=StreamTokenizer.TT_EOF){
           token = tokenizer.nextToken();
           switch(token){
              case StreamTokenizer.TT_NUMBER:
                  System.out.println("Número:");
                 break;
              case StreamTokenizer.TT_WORD:
                 System.out.println("Palavra:");
                 break;
               case StreamTokenizer.TT_EOL:
                  System.out.println("Quebra de linha");
                   break;
           }
            System.out.println(tokenizer.toString());
}
```

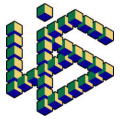
Persistência baseada em XML

- A partir do JDK 1.4, há o suporte a persistência de objetos através de arquivos XML.
- Podemos ver isto em XMLObject.java (que salva objetos em formato XML) e XMLObjectRead.java (que lê os objetos salvos pelo programa anterior)

Exemplos de gravação e leitura

- ObjetoParaDisco.java

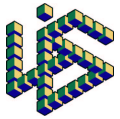
```
FileOutputStream out = new
  FileOutputStream("Agora.txt");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Agora é"); // escreve uma String
s.writeObject(new Date()); // escreve um objeto
s.flush();
```



Escrita de objetos em XML

- XMLObject.java

```
FileOutputStream fout = new
  FileOutputStream("DateOut.xml");
XMLEncoder enc = new XMLEncoder(fout);
Date d = new Date();
enc.writeObject(d);

Ponto p = new Ponto(5, 6);
enc.writeObject(p);
enc.close();
```



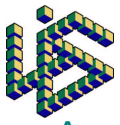
Exemplos de gravação e leitura

- ObjetoDoDisco.java

```
FileInputStream in = new
  FileInputStream("Agora.txt");
ObjectInputStream s = new
  ObjectInputStream(in);
String today =
  (String)s.readObject();
Date date = (Date)s.readObject();
```

Arquivos de acesso aleatório

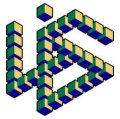
- Alguns métodos:
- getFilePointer()
- skipBytes(int n)
- seek(long pos)
- read<tipo>
- write<tipo>



Objetos em XML

- Arquivo salvo pelo programa do slide anterior

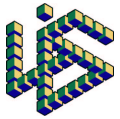
```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1" class="java.beans.XMLDecoder">
 <object class="java.util.Date">
  <long>1063156097556</long>
 </object>
 <object class="Ponto">
  <void property="x">
   <int>5</int>
  </void>
  <void property="y">
   <int>6</int>
  </void>
 </object>
</java>
```



Exemplo

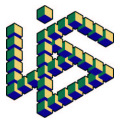
- AcessoAleatorio.java. Imprime um “Oi” de 10 em 10 posições no arquivo de entrada.

```
RandomAccessFile raf = new
  RandomAccessFile("xis.txt", "rw");
long comprimento = raf.length();
int curr = 1;
while (curr < comprimento){
    curr = curr + 10;
    raf.seek(curr);
    raf.writeBytes("Oi");
}
```



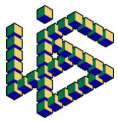
Arquivos de acesso aleatório

- Os arquivos de acesso aleatório não pertence à hierarquia de classes de InputStream, InputReader, OutputStream ou OutputWriter.
- Comporta-se como um grande *array* de bytes armazenado no sistema de arquivos, permitindo leitura e escrita.



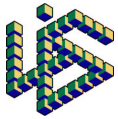
Exemplo que converte um arquivo para maiúsculas

```
BufferedReader bufferedReader = new BufferedReader(reader);

        String line = " ";
        while(line!=null){
            try {
                line = bufferedReader.readLine();
            } catch (IOException ioEx){
                System.out.println("Problemas na leitura do arquivo");
            }
            if (line!=null){
                System.out.println(line.toUpperCase());
            }
        }
    }
}
```



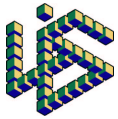
Decisões a tomar para fazer programas com I/O

- Meu dados são caracteres ou bytes?
- Preciso de processamento mais sofisticado? Tokenização, compactação?
- Preciso de acesso aleatório?



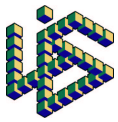
Finalizações

- Dúvidas? Mande e-mail para fabio.rmiranda@sp.senac.br
- Livro online http://www.mindview.net/Books/TIJ/ é muito bom
- http://developer.java.sun.com tem muito material bom para treinamento (grátis!!)



Exemplo que converte um arquivo para maiúsculas

```
import java.io.*;

public class FileLoad {

    public static void main(String[] args) {
        if (args.length < 1){
            System.out.println("Programa precisa de 1 argumento");
            return;
        }

        FileReader reader = null;
        try {
            reader = new FileReader(args[0]);
        }
        catch (FileNotFoundException ex) {
            System.out.println("Arquivo não encontrado: "+args[0]);
        }
```



Material online

- http://www.cei.sp.senac.br/~fabio.rmiranda