

Java First-Tier: Aplicações

Entrada & Saída

Grupo de Linguagens de Programação
 Departamento de Informática
 PUC-Rio

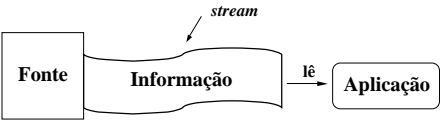
Motivação

- Uma aplicação normalmente precisa “obter” e/ou “enviar” informações a fontes/destinos externos
 - arquivos, conexões de rede, memória
- Essas informações podem ter vários tipos
 - bytes/caracteres, dados, objetos
- Java utiliza um mecanismo genérico que permite tratar E/S de forma uniforme
 - *Streams* de entrada e saída

2

Stream de Entrada

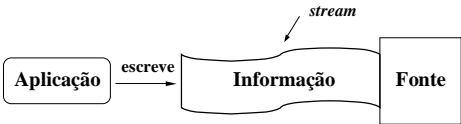
- Para obter informações, uma aplicação abre um *stream* de uma fonte (arquivo, *socket*, memória) e lê sequencialmente



3

Stream de Saída

- Para enviar informações, uma aplicação abre um *stream* para um destino (arquivo, *socket*, memória) e escreve sequencialmente



4

Leitura e Escrita de *Streams*

- Independentemente da fonte/destino e do tipo de informações, os algoritmos para leitura e escrita são basicamente os mesmos

<p>Leitura abre um <i>stream</i> enquanto há informação lê informação fecha o <i>stream</i></p>	<p>Escrita abre um <i>stream</i> enquanto há informação escreve informação fecha o <i>stream</i></p>
--	---

5

Pacote **java.io**

- Coleção de classes (*streams*) que suportam esses algoritmos
- As classes são divididas em duas hierarquias, baseadas no tipo de dados (*bytes* ou caracteres) sobre os quais operam
 - **InputStream/OutputStream**
 - **Reader/Writer**

} classes abstratas

6

Streams de Bytes

- As classes **InputStream** e **OutputStream** são as superclasses abstratas de todos os *streams* de bytes (dados binários)
 - **InputStream** define um método abstrato **read** para ler um byte de uma *stream*
 - **OutputStream** define um método abstrato **write** para escrever um byte em uma *stream*
- Subclasses provêem E/S especializada para cada tipo de fonte/destino

7

Exemplo: **System.in**

- É um objeto do tipo **InputStream**
- Esse *stream* já está aberto e pronto para prover dados à aplicação

```
public static final InputStream in

int bytesProntos = System.in.available();
if (bytesProntos > 0){
    byte[] entrada = new byte[bytesProntos];
    System.in.read(entrada);
}
```

8

Exemplo: **System.out**

- É um objeto do tipo **PrintStream**, subclasse de **OutputStream**
- Esse tipo de *stream* fornece a seu “destino” representações de vários tipos de dados

```
public static final PrintStream out

public void print(float f)
public void print(String s)
public void println(String s)
```

9

IOException

- É uma extensão da classe **Exception**
- Sinaliza a ocorrência de uma falha ou interrupção em uma operação de E/S
- Algumas subclasses:
 - **EOFException**, **FileNotFoundException**, **InterruptedIOException**, **MalformedURLException**, **SocketException**.

10

Streams de Caracteres

- As classes **Reader** e **Writer** são as superclasses abstratas de todos os *streams* de caracteres
 - **Reader** define um método abstrato **read** para ler uma sequência de caracteres de uma *stream*
 - **Writer** define um método abstrato **write** para escrever uma sequência de caracteres em uma *stream*
- Subclasses provêem E/S especializada diferentes tipos de fonte/destino

11

Streams de Strings

- **StringReader**

```
public StringReader(String str)
```
- **StringWriter**

```
public StringWriter(int buf_size)
public String toString()
```

12

Buffered Streams

- Por *default*, os *streams* não são *bufferizados*
 - essa funcionalidade pode ser obtida adicionando-se uma “camada” sobre o *stream*
- **BufferedInputStream, BufferedOutputStream**

```
public BufferedInputStream(InputStream in, int size)
```
- **BufferedReader, BufferedWriter**

```
public BufferedReader(Reader in, int size)
public String readLine() throws IOException
```

13

Streams de Conversão

- Pontes entre *streams* de *bytes* e de caracteres

```
public InputStreamReader(InputStream i)
public InputStreamReader(InputStream i, String enc)
    throws UnsupportedEncodingException
public OutputStreamWriter(OutputStream o)
public OutputStreamWriter(OutputStream o, String enc)
    throws UnsupportedEncodingException
```
- Para maior eficiência, pode-se utilizar *streams* *bufferizadas*:

```
BufferedReader in = new BufferedReader(new
    InputStreamReader(System.in));
```

14

Entrada/Saída em arquivos

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

15

Classe FileInputStream

- Especialização de **InputStream** para leitura de arquivos

```
public FileInputStream(String name) throws
    FileNotFoundException
public FileInputStream(File file) throws
    FileNotFoundException
```
- Usando *stream* *bufferizada*:

```
BufferedInputStream in =
    new BufferedInputStream
        (new FileInputStream("arquivo.dat"));
```

16

Classe FileOutputStream

- Especialização de **OutputStream** para escrita em arquivos

```
public FileOutputStream(String name) throws
    FileNotFoundException
public FileOutputStream(String name, boolean append)
    throws FileNotFoundException
public FileOutputStream(File file) throws
    FileNotFoundException
```

17

Classe FileReader

- É uma subclasse de **InputStreamReader**

```
public FileReader(String name) throws
    FileNotFoundException
public FileReader(File file) throws
    FileNotFoundException
```
- Usando *stream* *bufferizada*:

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

18

Exemplo de Leitura de Arquivo

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

19

Classe `FileWriter`

- É uma subclasse de `OutputStreamWriter`

```
public FileWriter(String name) throws
    IOException

public FileWriter(String name, boolean append)
    throws IOException

public FileWriter(File file) throws
    IOException
```

20

Exemplo de leitura e escrita

```
import java.io.*;
public class Copy {
    public static void main(String[] args) throws IOException {
        FileReader in = new FileReader("filein.txt");
        FileWriter out = new FileWriter("fileout.txt");
        int c;
        while ((c = in.read()) != -1)
            out.write(c);
        in.close();
        out.close();
    }
}
```

21

Classe `File`

- Representa um arquivo (ou diretório) no sistema de arquivos nativo
- Permite obter informações sobre arquivos e diretórios
- Permite também executar operações como criar, renomear e apagar arquivos e diretórios

22

Streams de Dados

- Definidos por interfaces
 - `DataInput`
 - `DataOutput`
- Permitem escrita e leitura de tipos básicos
- Essas interfaces são implementadas por
 - `DataInputStream`
 - `DataOutputStream`
 - `RandomAccessFile`

23

Exemplo de *stream* de dados

```
try {
    FileInputStream fin = new FileInputStream("arquivo.dat");
    DataInputStream din = new DataInputStream(fin);
    int num_valores = din.readInt();
    double[] valores = new double[num_valores];
    for (int i = 0; i < num_valores; i++)
        valores[i] = din.readDouble();
} catch (EOFException e) {
    ...
} catch (FileNotFoundException e) {
    ...
} catch (IOException e) {
    ...
}
```

24

Classe `RandomAccessFile`

- Permite a leitura e escrita em um arquivo de acesso randômico
- Implementa as interfaces `DataInput` e `DataOutput`
- Possui um *file pointer* que indica a posição (índice) corrente
 - o *file pointer* pode ser obtido através do método `getFilePointer` e alterado através do método `seek`

25

Exemplo

```
import java.io.*;
public class TesteRandom{
    public static void main(String argv[]){
        try {
            TesteRandom r = new TesteRandom();
            RandomAccessFile raf = new RandomAccessFile("teste.txt","rw");
            r.escreve(raf);
            r.leUm(raf,2); r.escreveUm(raf,2,'x'); r.leUm(raf,2);
        } catch(IOException ioe) {System.out.println(ioe);}
    }
    public void escreve(RandomAccessFile raf) throws IOException {
        char[] letras = {'a', 'b', 'c', 'd'};
        for(int i=0; i<4;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);
    }
}
```

26

Streams de Objetos

- Definidos pelas interfaces `ObjectInput` e `ObjectOutput`
 - implementadas por `ObjectInputStream` e `ObjectOutputStream`
- `ObjectInput` estende `DataInput` para incluir objetos, arrays e Strings
- `ObjectOutput` estende `DataOutput` para incluir objetos, arrays e Strings

27

Utilização de *streams* de Objetos

- Um `ObjectInputStream` “deserializa” dados e objetos anteriormente escritos através de um `ObjectOutputStream`.
- Cenários de utilização:
 - persistência de objetos, quando esses *streams* são usados em conjunto com `FileInputStream` e `FileOutputStream`
 - transferência de objetos entre *hosts*

28

`ObjectInputStream`

```
public final Object readObject()
    throws ClassNotFoundException,
    IOException
```

29

`ObjectOutputStream`

```
public final void writeObject(Object obj)
    throws IOException
```

30

Exemplo de serialização

```
FileOutputStream ostream = new
    FileOutputStream("t.tmp");
ObjectOutputStream p = new
    ObjectOutputStream(ostream);
p.writeInt(12345);
p.writeObject("Today");
p.writeObject(new Date());
p.flush();
ostream.close();
```

31

Recuperando os objetos

```
FileInputStream istream = new
    FileInputStream("t.tmp");
ObjectInputStream p = new
    ObjectInputStream(istream);
int i = p.readInt();
String today = (String)p.readObject();
Date date = (Date)p.readObject();
istream.close();
```

32

Interface **Serializable**

- Somente objetos cujas classes implementem a “marker interface” **Serializable** podem ser serializados
- Essa interface não tem métodos, mas uma classe “serializable” pode definir métodos **readObject** e **writeObject** para fazer validações no estado do objeto

33

Exemplo

```
class Funcionario implements Serializable {
    ...
    private void readObject(ObjectInputStream is)
        throws ClassNotFoundException, IOException
    {
        is.defaultReadObject();
        if (!isValid())
            throw new IOException("Invalid Object");
    }
    private boolean isValid() {
        ...
    }
}
```

34

Uniform Resource Locator

- A classe **URL** modela URLs, permitindo a obtenção de informações e conteúdo de páginas na Web
- Essa classe é parte do pacote **java.net**

35

Exemplo de Uso de URL

```
import java.io.*;
import java.net.*;

public class PegaPagina {
    public static void main(String[] args) throws Exception {
        if (args.length == 0) {
            System.err.println("Forneça o endereço da página.");
            return;
        }
        URL url = new URL(args[0]);
        InputStream is = url.openStream();
        Reader r = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(r);
        String l;
        while ((l = br.readLine()) != null) {
            System.out.println(l);
        }
    }
}
```

36

Lendo de Arquivos .jar

- A classe **Class** provê métodos para obter um recurso como **URL** ou **InputStream**. Quem efetivamente obtém o recurso é o *class loader* da classe em questão, que sabe de onde ela foi obtida

```
public URL getResource(String name)
public InputStream getResourceAsStream(String name)
```

37

Exemplos

- Exemplo do Applet

```
getAudioClip(getClass().getResource("spacemusic.au"));
```

- Outro exemplo

```
InputStream is =
    getClass().getResourceAsStream("arquivo.dat");
```

38