

Computação Orientada a Objetos

Arquivos Java

Slides baseados em:

- Deitel, H.M.; Deitel P.J. **Java: Como Programar**, Pearson Prentice Hall, 6a Edição, 2005.
- Deitel, H.M.; Deitel P.J. **Java: Como Programar**, Pearson
- Prentice Hall, 9a Edição, 2010.

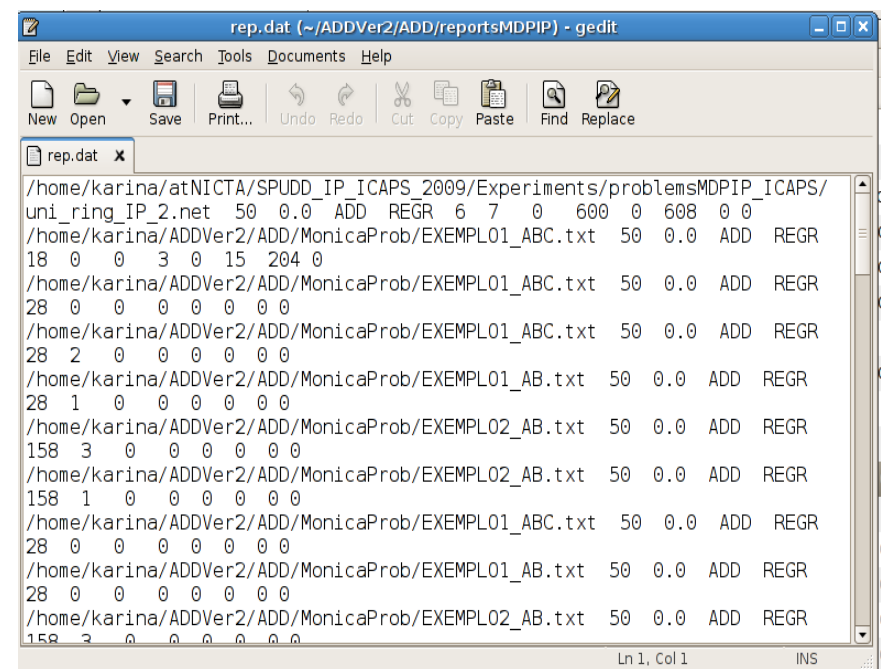
Profa. Karina Valdivia Delgado
EACH-USP

Introdução

- Programadores utilizam **arquivos** para:
 - armazenar dados a longo prazo.
- Dados armazenados em arquivos são chamados de **persistentes**:
 - eles existem mesmo depois que os programas que os criaram tenham terminado.
- O Java vê cada arquivo como um **fluxo**
- O termo **fluxo** se refere a dados que são lidos ou gravados em um arquivo.

Introdução

- Discutiremos:
 - Processamento de arquivos binários e arquivos de texto
 - Serialização de objetos
 - Processamento de arquivos de acesso aleatório



Arquivos e fluxos

- Um programa Java abre um arquivo criando e **associando um objeto ao fluxo** de bytes ou caracteres.
- Podem ser associados fluxos a diferentes dispositivos.
- Ex: `System.in`, `System.out` e `System.err` são objetos de fluxo que podem ser associados a diferentes dispositivos: tela, arquivo, etc.

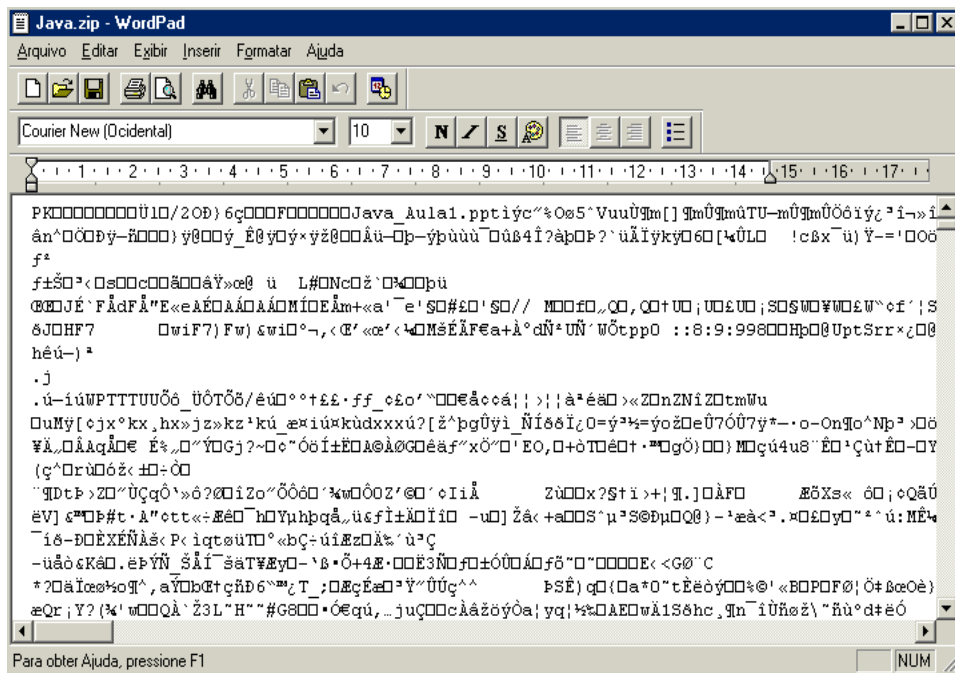
Arquivos e fluxos

- O processamento de arquivos é realizado utilizando o pacote **java.io**
- Esse pacote inclui definições para classes de fluxo.
- Algumas classes de fluxo são:
 - **FileInputStream**
 - **FileOutputStream**
 - **FileReader**
 - **FileWriter**

Arquivos e fluxos

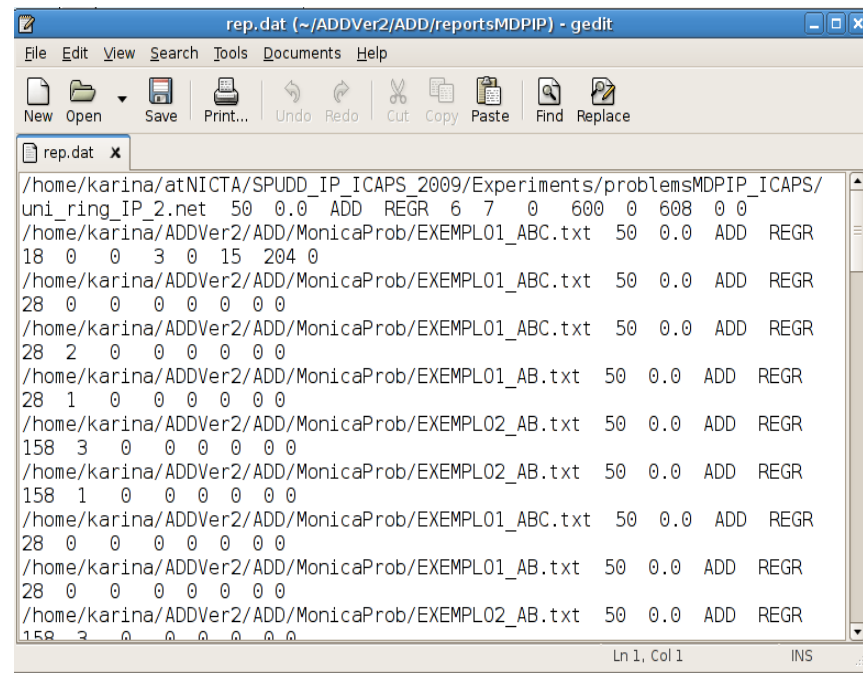
Arquivos binários

- **FileInputStream**: para entrada baseada em **bytes**;
- **FileOutputStream**: para saída baseada em **bytes**;



Arquivos de texto

- **FileReader**: para entrada baseada em **caracteres**;
- **FileWriter**: para saída baseada em **caracteres**.



Arquivos e fluxos

- Além dessas classes temos em **java.util**:
 - **Scanner**: para entrada baseada em **caracteres** a partir do teclado ou de um arquivo;
 - **Formatter**: para saída baseada em **caracteres** na tela ou em arquivo.

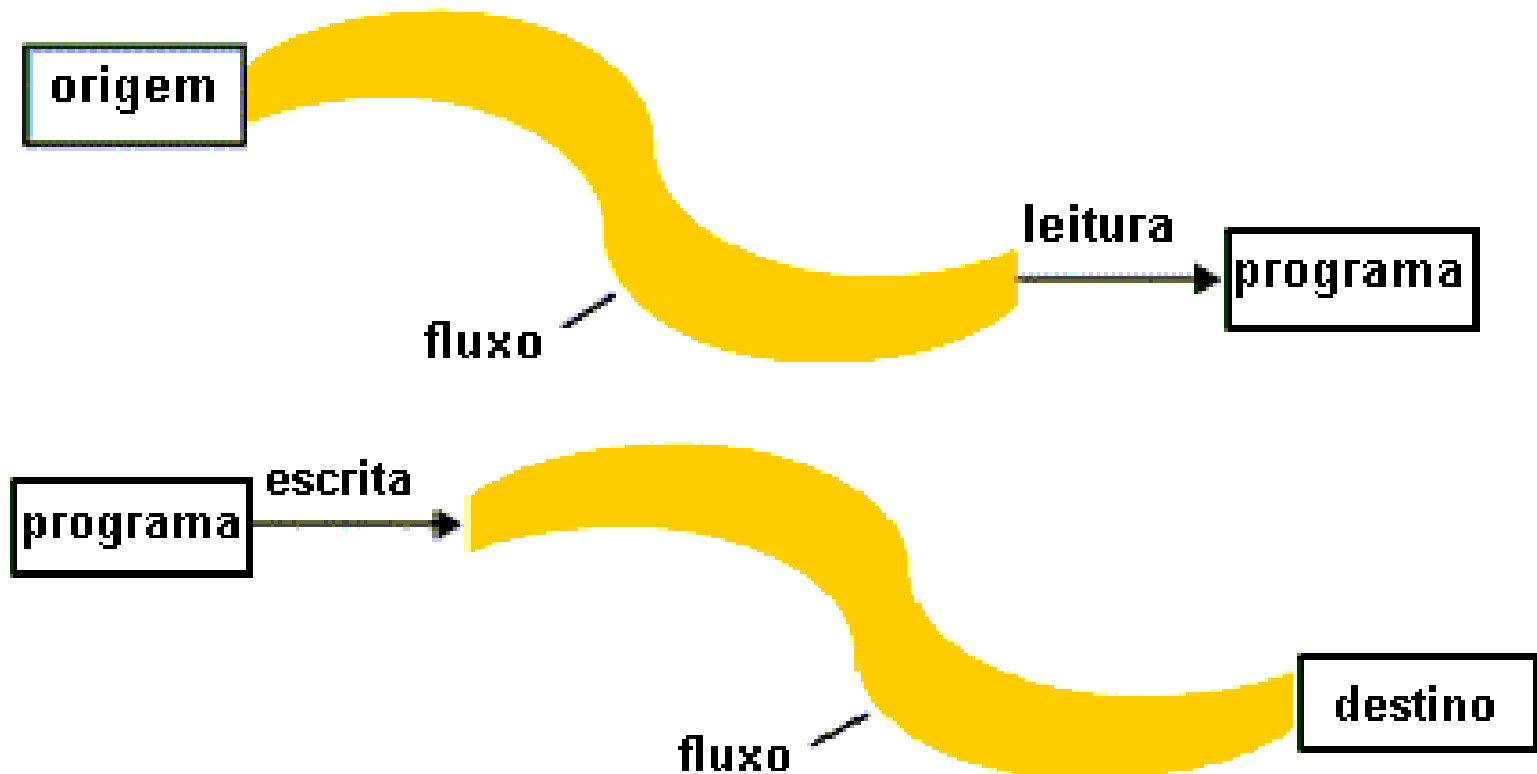
Arquivos e tipos de acesso

- Arquivos de acesso sequencial
- Arquivos de acesso aleatório

Arquivos de acesso sequencial

Arquivos de acesso sequencial

- Vistos como um fluxo sequencial de caracteres ou bytes.
- Só podem ser percorridos do início para o fim (e não no sentido contrário, nem de forma aleatória).



Procedimento geral

Leitura

abrir fluxo

**enquanto houver dados
ler**

fechar fluxo

Gravação

abrir fluxo

**enquanto houver dados
escrever**

fechar fluxo

Fluxos de bytes

Fluxos de bytes

- Todos os fluxos de bytes são uma subclasse de **InputStream** ou **OutputStream** (classes abstratas).
- Utilizados para manipulação de arquivos binários (ex: som, imagem ou dados em geral).

Fluxos de bytes: leitura

- Para ler um byte de um arquivo usamos o leitor de arquivo **FileInputStream** (classe concreta de **InputStream**)
- Um fluxo de entrada de bytes pode ser criado com o construtor

FileInputStream(String nome)

O argumento deverá ser o nome do arquivo o qual pode incluir o caminho absoluto

- Ex:

```
InputStream fluxo =  
    new FileInputStream ("arquivo.dat");
```

Fluxos de bytes: escrita

- Um fluxo de saída de bytes pode ser criado com o construtor

FileOutputStream(String nome)

- Ex:

```
OutputStream fluxo =  
    new FileOutputStream ("arquivo.dat");
```


Serialização de objetos

Serialização de objetos

- Java permite ler e escrever objetos inteiros em arquivos
- Realizada com fluxos de bytes (arquivos binários)
- A Classe de dados deve implementar a interface **Serializable**

Serialização de objetos

- **Serialização de objetos:** mecanismo para ler ou gravar um objeto inteiro a partir de um arquivo
- Um **objeto serializado** é um objeto representado como uma sequência de **bytes** que inclui:
 - dados do objeto
 - as informações sobre o tipo do objeto;
 - os tipos dos dados armazenados no objeto.
- O objeto pode ser **desserializado** a partir do arquivo

Serialização de objetos

- As classe `ObjectInputStream` e `ObjectOutputStream` permitem que objetos sejam lidos ou gravados em um **fluxo**
- Para usar a serialização com arquivos inicializamos esses objetos de fluxo com objetos de fluxo que lêem e gravam arquivos:
 - `FileInputStream` e `FileOutputStream`
- A inicialização de objetos de fluxo com outros objetos de fluxo é chamado de **empacotamento**.

Serialização de objetos

- Abertura do fluxo para leitura

```
ObjectInputStream entrada =  
    new ObjectInputStream(  
        new FileInputStream("meusobjetos.ser"));
```

- Abertura do fluxo para escrita

```
ObjectOutputStream saida =  
    new ObjectOutputStream(  
        new FileOutputStream("meusobjetos.ser"));
```

Leitura, escrita e fechamento

- Leitura de dados

objeto = (Tipo) entrada.readObject();

- Escrita de dados

saida.writeObject(objeto);

- Fechamento do arquivo

entrada.close();

saida.close();

Serialização - Exemplo

- Considere uma classe representando registros de itens (produtos) em um estoque:

```
import java.io.Serializable;

public class Produto implements Serializable {
    private String nome;
    private int unidades;    // estoque em unidades
    private float custo;    // custo unitário
    public Produto(){
        this(" ", 0 , 0.0);
    }
    ...
}
```

Serialização - Exemplo

- Considere uma classe representando registros de itens (produtos) em

A interface Serializable é uma interface de marcação (não contém nenhum método)

```
import java.io.Serializable;

public class Produto implements Serializable {
    private String nome;
    private int unidades; // estoque em unidades
    private float custo; // custo unitário
    public Produto(){
        this(" ", 0 , 0.0);
    }
    ...
}
```


Serialização - Exemplo

- Considere
itens em
quantidade

A classe é marcada para permitir que seus objetos sejam “Serializable”.
Devemos verificar que cada variável da instância da classe também seja “Serializable”

```
import java.io.Serializable;

public class Produto implements Serializable {
    private String nome;
    private int unidades; // estoque em unidades
    private float custo; // custo unitário
    public Produto(){
        this(" ", 0 , 0.0);
    }
    ...
}
```

Serialização - Exemplo

- Escrevendo em um fluxo de objetos

```
Produto item = new Produto("livro java", 10, 148.50);
try {
    FileOutputStream fluxo = new FileOutputStream("item.ser");
    ObjectOutputStream objarq = new ObjectOutputStream(fluxo);
    objarq.writeObject(item);
    objarq.close();
}
catch(IOException ioExc) {
    System.out.println(ioExc.getMessage());
    ioExc.printStackTrace();
}
...
```

Serialização

■ Escrita

Pode lançar uma **IOException**, problema ao abrir o arquivo:

- arquivo aberto para gravação em uma unidade com espaço insuficiente,

Produto item;

```
try {  
    FileOutputStream fluxo = new FileOutputStream("item.ser");  
    ObjectOutputStream objarq = new ObjectOutputStream(fluxo);  
    objarq.writeObject(item);  
    objarq.close();  
}  
catch(IOException ioExc) {  
    System.out.println(ioExc.getMessage());  
    ioExc.printStackTrace();  
}  
...
```

Serialização - Exemplo

- Escrevendo em um fluxo de objetos

```
Produto item = new Produto("1");
try {
    FileOutputStream fluxo = new FileOutputStream("item.ser");
    ObjectOutputStream objarq = new ObjectOutputStream(fluxo);
    objarq.writeObject(item);
    objarq.close();
}
catch(IOException ioExc) {
    System.out.println(ioExc.getMessage());
    ioExc.printStackTrace();
}
...
```

Uma instrução para gravar o objeto inteiro!!!

Serialização - Exemplo

■ Escrevendo

Pode lançar uma **IOException**, problema ao escrever no arquivo:

- unidade com **espaço** insuficiente

```
Produto item = new Produto();
try {
    FileOutputStream fluxo = new FileOutputStream("item.ser");
    ObjectOutputStream objjarq = new ObjectOutputStream(fluxo);
    objjarq.writeObject(item);
    objjarq.close();
}
catch(IOException ioExc) {
    System.out.println(ioExc.getMessage());
    ioExc.printStackTrace();
}
...
```

Serialização - Exemplo

- Escrevendo em um arquivo

Fecha o objeto ObjectOutputStream e o objeto FileOutputStream

```
Produto item = new Produto(1, "Arroz", 1.2);
try {
    FileOutputStream fluxo = new FileOutputStream("item.ser");
    ObjectOutputStream objarq = new ObjectOutputStream(fluxo);
    objarq.writeObject(item);
    objarq.close();
}
catch(IOException ioExc) {
    System.out.println(ioExc.getMessage());
    ioExc.printStackTrace();
}
...
```

Serialização - Exemplo

- Escrevendo em um fluxo de objetos

```
Produto item = new Produto();
try {
    FileOutputStream fluxo = new FileOutputStream("arquivo.dat");
    ObjectOutputStream objarq = new ObjectOutputStream(fluxo);
    objarq.writeObject(item);
    objarq.close();
}
catch(IOException ioExc) {
    System.out.println(ioExc.getMessage());
    ioExc.printStackTrace();
}
...
```

close pode lançar uma **IOException** quando o arquivo não pode ser fechado adequadamente

Serialização - Exemplo

- Lendo a partir de um fluxo de objetos

```
...  
Produto item1;  
try {  
    FileInputStream fluxo = new FileInputStream("item.ser");  
    ObjectInputStream objarq = new ObjectInputStream(fluxo);  
    item1 = (Produto) objarq.readObject();  
    System.out.println(item1);  
    objarq.close();  
}  
catch (FileNotFoundException e) {  
    System.out.println("Arquivo não encontrado");  
}  
catch (IOException ioExc) {  
    System.out.println(ioExc.getMessage());  
    ioExc.printStackTrace();  
} ...  
...
```


Serialização - Exemplo

- Lendo a partir de um fluxo de objetos

Pode lançar uma **FileNotFoundException**, se o arquivo não existe.

```
...
Produto item;
try {
    FileInputStream fluxo = new FileInputStream("item.ser");
    ObjectInputStream objarq = new ObjectInputStream(fluxo);
    item1 = (Produto) objarq.readObject();
    System.out.println(item1);
    objarq.close();
}
catch (FileNotFoundException e) {
    System.out.println("Arquivo não encontrado");
}
catch (IOException ioExc) {
    System.out.println(ioExc.getMessage());
    ioExc.printStackTrace();
} ...
```

Serialização - Exemplo

■ Lendo

Pode lançar uma **IOException**

```
...  
Produto item;  
try {  
    FileInputStream fis = new FileInputStream("item.ser");  
    ObjectInputStream objarq = new ObjectInputStream(fis);  
    item1 = (Produto) objarq.readObject();  
    System.out.println(item1);  
    objarq.close();  
}  
catch (FileNotFoundException e) {  
    System.out.println("Arquivo não encontrado");  
}  
catch (IOException ioExc) {  
    System.out.println(ioExc.getMessage());  
    ioExc.printStackTrace();  
} ...
```

Serialização - Exemplo

- Lendo a partir de um fluxo de objetos

```
...  
Produto item1;  
try {  
    FileInputStream fluxo = new  
    ObjectInputStream objarq = new ObjectInputStream(fluxo),  
    item1 = (Produto) objarq.readObject();  
    System.out.println(item1);  
    objarq.close();  
}  
catch (FileNotFoundException e) {  
    System.out.println("Arquivo não encontrado");  
}  
catch (IOException ioExc) {  
    System.out.println(ioExc.getMessage());  
    ioExc.printStackTrace();  
}
```

readObject devolve um objeto do tipo Object.

...

Serialização - Exemplo

- Lendo a partir de um fluxo de objetos

```
...
Produto item1;
try {
    FileInputStream fluxo = new
    ObjectInputStream objarq = new
    item1 = (Produto) objarq.readObject();
    System.out.println(item1);
    objarq.close();
}
catch (FileNotFoundException e) {
    System.out.println("Arquivo não encontrado");
}
catch(IOException ioExc) {
    System.out.println(ioExc.getMessage());
    ioExc.printStackTrace();
}...
```

Pode lançar outras exceções:
EOFException,
ClassNotFoundException.

Serialização - Exemplo

- Lendo mais de um objeto a partir de um fluxo de objetos, suponha que o arquivo está aberto

```
public void readRecords(){
    Produto item1;
    try {
        while (true){
            item1 = (Produto) objarq.readObject();
            System.out.println(item1);
        }
    }
    catch(IOException endOfFileExc) {
        System.err.println("Fim do arquivo");
        return;
    }
    catch(ClassNotFoundException classNotFoundExc) {
        System.err.println("Não foi possível criar o objeto");
    } ...
}
```

Serialização - Exemplo

- Lendo mais de um objeto a partir de um fluxo de objetos, suponha que o arquivo está aberto

```
public void readRecords(){
    Produto item1;
    try {
        while (true){
            item1 = (Produto) objarq.readObject();
            System.out.println(item1);
        }
    }
    catch(IOException endOfFileExc) {
        System.err.println("Fim do arquivo");
        return;
    }
    catch(ClassNotFoundException classNotFoundExc) {
        System.err.println("Não foi possível criar o objeto");
    } ...
}
```

se existir uma tentativa de leitura depois do final do arquivo, readObject lança uma **EOFException**

Serialização - Exemplo

- Lendo mais de um objeto a partir de um fluxo de objetos, suponha que o arquivo está aberto

```
public void readRecords(){
    Produto item1;
    try {
        while (true){
            item1 = (Produto) objarq.readObject();
            System.out.println(item1);
        }
    }
    catch(IOException endOfFileExc)
        System.err.println("Fim do arquivo");
    return;
}
catch(ClassNotFoundException classNotFoundExc) {
    System.err.println("Não foi possível criar o objeto");
} ...
}
```

se a classe do objeto sendo lido não puder ser localizada, readObject lança uma exceção

ClassNotFoundException

Serialização de coleções

- Estruturas de dados do tipo *Collection* podem ser lidas ou escritas na sua totalidade sem necessidade de iteração.

```
Set <Integer> s = new HashSet <Integer> ();  
  
...  
ObjectOutputStream saida =  
    new ObjectOutputStream(  
        new FileOutputStream(arquivo));  
saida.writeObject(s);
```


Serialização de coleções

```
TreeMap<Integer, String > mapa = new TreeMap<Integer, String>();
mapa.put(455, "vermelho");
mapa.put(333, "branco");
mapa.put(678, "amarelo");
mapa.put(455, "azul");
try {
    FileOutputStream fluxoOut=new FileOutputStream("myFile.ser");
    ObjectOutputStream fOut=new ObjectOutputStream(fluxoOut);
```

Serialização de coleções

```
TreeMap<Integer, String > mapa = new TreeMap<Integer, String>();
mapa.put(455, "vermelho");
mapa.put(333, "branco");
mapa.put(678, "amarelo");
mapa.put(455, "azul");
try {
    FileOutputStream fluxoOut=new FileOutputStream("myFile.ser");
    ObjectOutputStream fOut=new ObjectOutputStream(fluxoOut);
    fOut.writeObject(mapa);
    FileInputStream fluxoIn = new FileInputStream("myFile.ser");
    ObjectInputStream fIn=new ObjectInputStream(fluxoIn);
    TreeMap<Integer, String > mapaNovo=(TreeMap)fIn.readObject();
    fIn.close();
    fOut.close();
    System.out.println(mapaNovo);
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();}
```

Fluxos de caracteres

Fluxos de caracteres

- Usados para lidar com qualquer texto que seja representado pelo conjunto de caracteres Unicode de 16 bits. Ex:
`00000000 01001010` é o unicode de J
- As classes usadas para ler e escrever fluxos de caracteres são todas derivadas das classes **Reader** e **Writer**.

Fluxos de caracteres: leitura

- **FileReader** é a classe principal usada para a leitura de fluxos de caracteres em um arquivo.
- Um fluxo de entrada de caracteres é associado a um nome de arquivo usando o construtor

FileReader (String nome)

- Ex:

```
FileReader fluxo =  
    new FileReader ("index.txt");
```

Fluxos de caracteres: escrita

- A classe **FileWriter** é a classe usada para gravar um fluxo de caracteres em um arquivo.
- Pode ser criado com o construtor

FileWriter (String nome)

- Ex:

```
FileWriter fluxo =  
    new FileWriter ("index.txt");
```

Exemplo FileReader (com bloco finally)

```
int i;
FileReader entrada = null;
try{ entrada = new FileReader("exemplo.txt");
    while (true){
        i = entrada.read();
        if ( i == -1 ) break;
        char c = (char) i;
        System.out.print( c );    }
}
catch (FileNotFoundException e){
    System.out.println("Arquivo não encontrado"); }
catch (IOException e){
    System.err.println(e); }
finally{
    try{
        if (entrada != null){
            System.out.println("Fechando FileReader");
            entrada.close();
        }
        else
            System.out.println("O arquivo nao foi aberto");
    }
    catch (IOException e){
        System.out.println("O arquivo nao pode ser fechado");
    }
} //fim finally ...
```

Exemplo FileReader (com bloco finally)

```
int i;
FileReader entrada = null;
try{ entrada = new FileReader("exemplo.txt");
    while (true){
        i = entrada.read();
        if ( i == -1 ) break;
        char c = (char) i;
        System.out.print( c );
    }
} catch (FileNotFoundException e){
    System.out.println("Arquivo não encontrado"); }
catch (IOException e){
    System.err.println(e); }
finally{
    try{
        if (entrada != null){
            System.out.println("Fechando FileReader");
            entrada.close();
        }
        else
            System.out.println("O arquivo nao foi aberto");
    }
    catch (IOException e){
        System.out.println("O arquivo nao pode ser fechado");
    }
} //fim finally ...
```

-1 indica final de arquivo de caracteres

????

Gostaria de usar um fluxo de caracteres
sem ter que ler caractere a
caractere !!!

Fluxos de caracteres: leitura

- Podemos usar **FileReader** com um filtro de fluxo **BufferedReader**.
- Sintaxe:

BufferedReader br=

new **BufferedReader**(new **FileReader**(nomeArquivo));

- Um dos métodos mais úteis da classe **BufferedReader** permite ler uma linha de texto:

readLine()

Fluxos de caracteres: leitura

```
....  
...  
try {  
    FileReader f = new FileReader("entrada.txt");  
    BufferedReader in = new BufferedReader(f);  
    String linha = in.readLine();  
    while(linha != null ){  
        System.out.println(linha);  
        linha = in.readLine();  
    }  
  
    in.close();  
}  
  
catch (IOException e){  
    System.err.println(e).  
}  
...
```

O programa mostra na tela
linha por linha o arquivo
entrada.txt

Fluxos de caracteres: escrita

- A classe **PrintWriter** dispõe de métodos já conhecidos: **print** e **println** para cada um dos tipos primitivos e tipo String.
- Sintaxe:

PrintWriter pw =

new **PrintWriter** (new **FileWriter**(nomeArquivo));

Fluxos de caracteres:

- Além dessas classes temos em `java.util`:
 - **Scanner**: para entrada baseada em `caracteres` a partir do teclado ou de um arquivo;
 - **Formatter**: para saída baseada em `caracteres` na tela ou em arquivo, essa classe permite produzir saídas com formatação, com capacidade semelhante ao `printf`.

Exemplo Formatter

```
public class AccountRecord
{
    private int account;
    private String firstName;
    private String lastName;
    private double balance;
    ...
}
```

Exemplo Formatter

```
public class CreateTextFile
{
    private Formatter output;
    public void openFile()
    {
        try
        {
            output = new Formatter( "clients.txt" );
        }
        catch ( SecurityException securityException )
        {
            System.err.println( "You do not have write access to this file." );
            System.exit( 1 ); // terminate the program
        } // end catch
        catch ( FileNotFoundException fileNotFoundException )
        {
            System.err.println( "Error opening or creating file." );
            System.exit( 1 ); // terminate the program
        } // end catch
    } // end method openFile
}
```

Exemplo Formatter

```
public void addRecords()
{
    AccountRecord record = new AccountRecord();
    Scanner input = new Scanner( System.in );
    System.out.println( "To terminate input, type the end-of-file indicator " );
    System.out.printf( "%s\n%s", "Enter account number > 0, first and last name and balance.", "?" );
    while( input.hasNext())// loop until end-of-file indicator
    {
        try // output values to file
        {
            record.setAccount( input.nextInt() );
            record.setFirstName( input.next() );
            record.setLastName( input.next() );
            record.setBalance( input.nextDouble() );
            if ( record.getAccount() > 0 )
            {
                output.format( "%d %s %s %.2f\n", record.getAccount(),
                                record.getFirstName(), record.getLastName(),
                                record.getBalance() );
            } // end if
            else
            {System.out.println( "Account number must be greater than 0." );}
        } // end try
    }
}
```


Exemplo Formatter

```
catch ( FormatterClosedException formatterClosedException )
{
    System.err.println( "Error writing to file." );
    return;
} // end catch
catch ( NoSuchElementException elementException )
{
    System.err.println( "Invalid input. Please try again." );
    input.nextLine(); // discard input so user can try again
} // end catch
System.out.printf( "%s\n%s", "Enter account number > 0,
                    first and last name and balance.", "?" );
} // end while
} // end method addRecords
public void closeFile()
{
    if ( output != null )
        output.close();
} // end method closeFile
} // end class CreateTextFile
```

Exemplo Scanner

```
public class ReadTextFile
{
    private Scanner input;
    public void openFile()
    {
        try
        {
            input = new Scanner( new File( "clients.txt" ) );
        } // end try
        catch ( FileNotFoundException fileNotFoundException )
        {
            System.err.println( "Error opening file." );
            System.exit( 1 );
        } // end catch
    } // end method openFile
}
```

Exemplo Scanner

```
public void readRecords()
{
    AccountRecord record = new AccountRecord();
    try // read records from file using Scanner object
    {
        while(input.hasNext() )
        {
            record.setAccount( input.nextInt() ); // read account number
            record.setFirstName( input.next() ); // read first name
            record.setLastName( input.next() ); // read last name
            record.setBalance( input.nextDouble() ); // read balance
            System.out.printf( "%-10d%-12s%-12s%10.2f\n", record.getAccount(),
                                record.getFirstName(),record.getLastName(),
                                record.getBalance() );
        } // end while
    } // end try
}
```

Exemplo Scanner

```
catch ( NoSuchElementException elementException )
{
    System.err.println( "File improperly formed." );
    input.close();
    System.exit( 1 );
} // end catch
catch ( IllegalStateException stateException )
{
    System.err.println( "Error reading from file." );
    System.exit( 1 );
} // end catch
} // end method readRecords

public void closeFile()
{
    if ( input != null )
        input.close();
} // end method closeFile
} // end class ReadTextFile
```

Arquivos de acesso aleatório

Arquivos de Acesso Aleatório

- Permitem ler ou escrever a partir de qualquer posição no arquivo
 - acesso rápido
- Podem ser criados utilizando a classe **RandomAccessFile**
 - permite que se trabalhe nos modos “leitura” (r), “gravação” (w).

Abertura do arquivo de Acesso Aleatório

- Para leitura

```
RandomAccessFile entrada = new  
RandomAccessFile("dados.bin", "r");
```

- Para escrita

```
RandomAccessFile saida = new  
RandomAccessFile("dados.bin", "w");
```

Arquivos de Acesso Aleatório

- Os registros podem ser acessados diretamente, por meio de um ponteiro (**ponteiro de arquivo**).
- Ao associar um **RandomAccessFile** a um arquivo:
 - Dados são lidos/escritos na posição do ponteiro de arquivo.
 - Todos os dados são tratados como tipos primitivos(formato binário)
 - int: 4 bytes;
 - double: 8 bytes;
 - etc

O ponteiro de arquivo

- Leitura e escrita ocorrem na posição do *ponteiro de arquivo* (iniciada em zero).
- Posição atual do ponteiro de arquivo:
`long posição = fluxo.getFilePointer();`
- Deslocamento para posição específica:
`fluxo.seek(posição);`
- Avanço de n posições (em bytes):
`fluxo.skipBytes(n);`

Arquivos de Acesso Aleatório

- Leitura (pode gerar *EOFException*)

```
int i = entrada.readInt();  
double d = entrada.readDouble();  
char c = entrada.readChar();  
String s = entrada.readUTF();
```

- Escrita

```
saida.writeInt(i);  
saida.writeDouble(d);  
saida.writeChar(c);  
saida.writeUTF(s);
```

- Fechamento

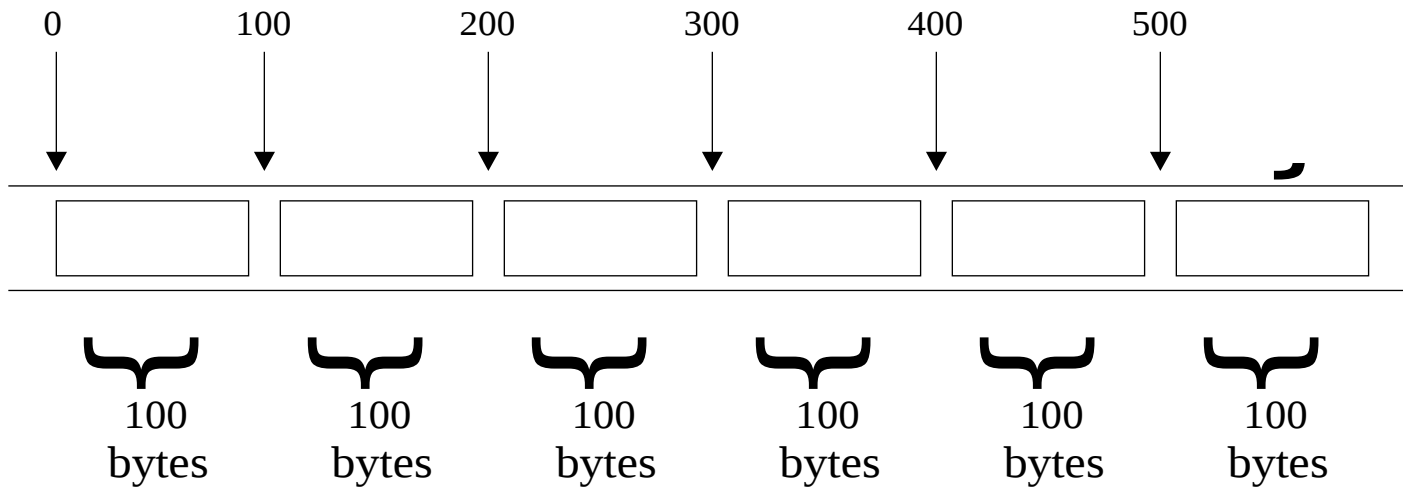
```
entrada.close();           saida.close();
```

Exemplo: registros bancários

- Considere uma classe representando registros bancários (nro.de conta corrente, nome e saldo).
- Deseja-se armazenar 100 registros em um arquivo de acesso aleatório:
 - contas numeradas de 00 a 99;

Exemplo: registros bancários

- Podemos organizar a informação de forma simples usando registros de tamanho fixo
 - Facilita o cálculo da localização exata de qualquer registro em relação ao início do arquivo.



Exemplo: registros bancários

- A classe Registro

```
public class Registro {  
    int nroconta;  
    String nome;  
    double saldo;  
}
```

Exemplo: registros bancários

- A classe Registro

```
public class Registro {  
    int nroconta;  
    String nome;  
    double saldo;  
}
```

4 bytes

8 bytes

????

Exemplo: calculando o tamanho de um registro

- Soma dos tamanhos de seus campos:
 - Tipo *char* possui tamanho 2
 - Tipo *String* possui tamanho pré-definido pelo programador (nro. caracteres * 2)
 - Tipos *int*, *double* etc têm seu tamanho (em bits) dado pela constante **SIZE**. **Exemplo:**
 $\text{TamanhoInteger} = \text{Integer.SIZE} / 8;$
 $\text{TamanhoDouble} = \text{Double.SIZE} / 8;$

Exemplo: registros bancários

- A classe Registro

```
public class Registro {  
    int nroconta;  
    String nome;  
    double saldo;  
}
```

4 bytes

8 bytes

Número de
caracteres * 2
bytes

Exemplo: registros bancários

- Método **tamanho()** da classe Registro:

```
public int tamanho () {  
    return((Integer.SIZE/8) +  
           15 * 2 +(Double.SIZE/8));  
    // ou simplesmente return(4+30+8);  
}
```

Exemplo: registros bancários

- Uma vez localizado o registro desejado, este pode ser:
 - reescrito com os métodos *writeInt*, *writeChar*, etc.
 - lido com os métodos *readInt*, *readChar*, etc.

Exemplo: registros bancários

- Criaremos na própria classe Registro, os métodos:
 - **escrever(arquivo)** : para escrever um registro na posição atual do arquivo especificado.
 - **ler(arquivo)** : para ler um registro da posição atual do arquivo especificado.

Exemplo: registros bancários

- Escrita de um registro:

```
public void escrever(RandomAccessFile f) throws IOException {  
    .....  
    // lê os dados e armazena nas variáveis de  
    instância  
    .....  
    f.writeInt(nroconta);  
    StringBuffer b = new StringBuffer(nome);  
    b.setLength( 15 );  
    f.writeUTF( b.toString() );  
    f.writeDouble(saldo);  
}
```

Cria e manipula strings modificáveis. Podemos usar `StringBuilder` no lugar de `StringBuffer`.

Exemplo: registros bancários

- Escrita de um registro:

```
public void escrever(RandomAccessFile f) throws IOException {  
    .....  
    // lê os dados e armazena nas variáveis de  
    instância  
    .....  
    f.writeInt(nroconta);  
    StringBuffer b = new StringBuffer(nome);  
    b.setLength( 15 );  
    f.writeUTF( b.toString() );  
    f.writeDouble(saldo);  
}
```

Usamos **setLength** para definir um tamanho fixo para qualquer nome. Se o nome tiver menos de 15 caracteres, os caracteres extras terão o valor '\0'

Exemplo: registros bancários

- Leitura de um registro:

```
public void ler(RandomAccessFile f) throws IOException
{
    nroconta = f.readInt();
    char letras[] = new char [15];
    for(int i=0;i<15;i++)
        letras[i] = f.readChar();
    nome = new String( letras ).replace( '\\0', ' ' );
    saldo = f.readDouble();
}
```