



Computação Orientada a Objetos

Prof. Marcos L. Chaim

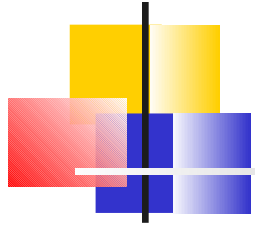
Parte 10 – Arquivos

Slides elaborados pela Profa. Patrícia R. Oliveira



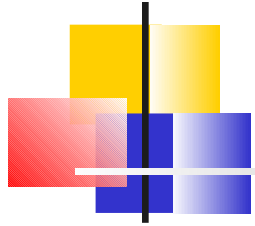
Introdução

- 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 termo fluxo se refere a dados que são lidos ou gravados em um arquivo



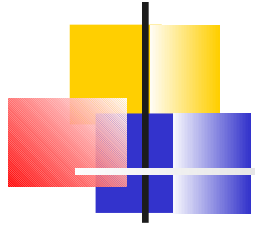
Hierarquia de dados

- Bit
 - menor item de dados em um computador
 - assume valor 0 ou 1
 - inadequado para leitor humano
 - usamos caracteres (dígitos, símbolos, letras)
 - representados no computador como padrões de 0's e 1's
 - em Java: caracteres Unicode compostos de dois bytes



Hierarquia de dados

- Byte
 - Grupo de 8 bits;
 - 2 bytes (16 bits) são usados para representar um caractere Unicode;
- Campo
 - Grupo de caracteres com significado;
 - Exemplo: endereço de funcionário;
- Registro
 - Grupo de campos relacionados;
 - Exemplo: registro de um funcionário.

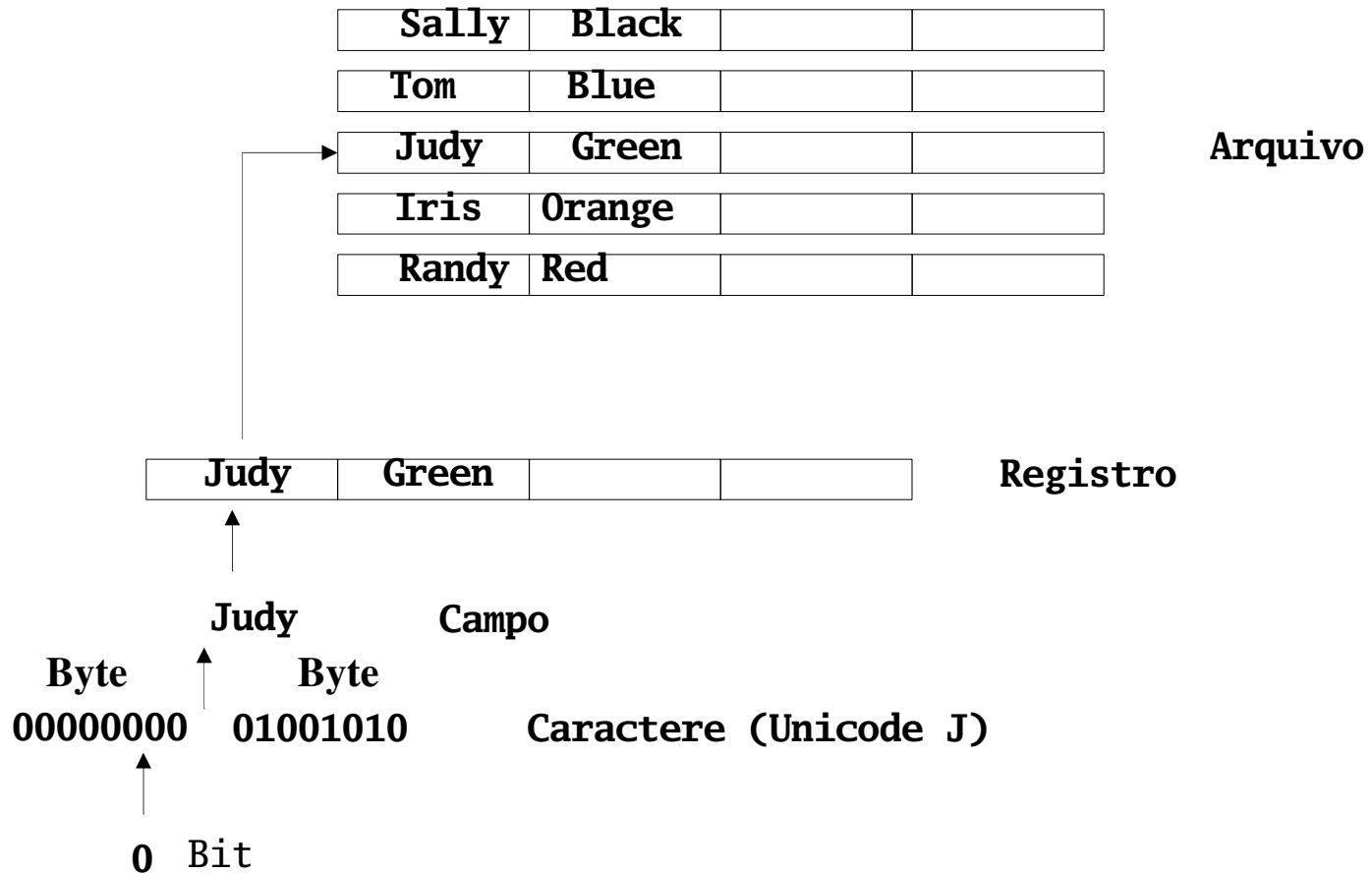


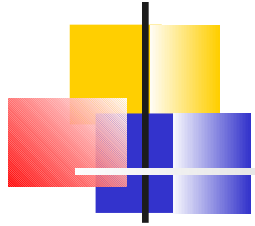
Hierarquia de dados

- Arquivo
 - Grupo de registros relacionados;
 - Exemplo: informações sobre muitos funcionários;
- Banco de Dados
 - Grupo de arquivos relacionados;
 - Exemplo: arquivo de folha de pagamento, arquivo de contas a receber, arquivos de contas a pagar, etc.



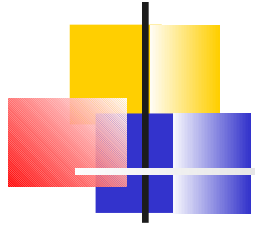
Exemplo





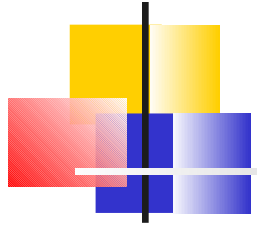
Arquivos e fluxos

- Java vê cada arquivo como um fluxo seqüencial de bytes
 - geralmente terminam com uma marca de final de arquivo ou um código especial
- Fluxos de arquivos podem ser utilizados para entrada e saída de dados como caracteres ou bytes



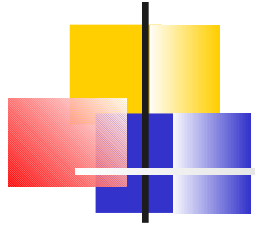
Arquivos e fluxos

- Arquivos binários: criados com base em fluxos de bytes
 - lidos por um programa que converte os dados em um formato legível por humanos
- Arquivos de texto: criados com base em fluxos de caracteres
 - podem ser lidos por editores de texto
- Um programa Java abre um arquivo criando e associando um objeto a um fluxo de bytes ou caracteres



Arquivos e fluxos

- O processamento de arquivos é realizado utilizando o pacote **java.io**
- Esse pacote inclui definições para classes de fluxo, como:
 - **FileInputStream:** para entrada baseada em bytes
 - **FileOutputStream:** para saída baseada em bytes



Arquivos e fluxos

- **FileReader:** para entrada baseada em caracteres
- **FileWrite:** para saída baseada em caracteres
- Obs: Essas classes herdam das classes **InputStream**, **OutputStream**, **Reader** e **Writer**, respectivamente



A classe *File*

- Útil para recuperar informações sobre arquivos e diretórios em disco
- Não abre nem processa arquivos
- É utilizada com objetos de outras classes do pacote **java.io** para especificar arquivos ou diretórios a manipular



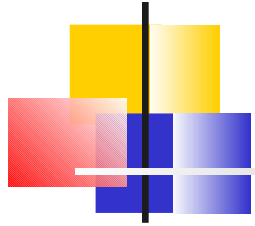
Criando objetos *File*

- **File(String nome)**
 - especifica o **nome** de um arquivo ou diretório para associar a um objeto **File**
 - o **nome** pode conter informações de caminho
 - caminho absoluto: inicia no diretório raiz e inclui todo o caminho levando ao arquivo
 - caminho relativo: inicia no diretório onde a aplicação foi iniciada



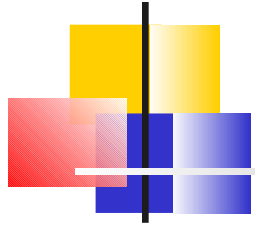
Métodos *File*

- **boolean canRead()**
 - retorna **true** se um arquivo puder ser lido pelo aplicativo
- **boolean canWrite()**
 - retorna **true** se um arquivo puder ser gravado pelo aplicativo
- **boolean exists()**
 - retorna **true** se o argumento para o construtor é um arquivo ou diretório válido



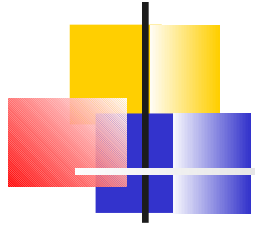
Métodos *File*

- **boolean isFile()**
 - retorna **true** se o nome especificado como argumento para o construtor é um arquivo
- **boolean isDirectory()**
 - retorna **true** se o nome especificado como argumento para o construtor é um diretório
- **boolean isAbsolute()**
 - retorna **true** se o nome especificado como argumento para o construtor é um caminho absoluto



Métodos *File*

- **String getAbsolutePath()**
 - retorna uma string com o caminho absoluto do arquivo ou diretório
- **String getName()**
 - retorna uma string com o nome do arquivo ou diretório
- **String getPath()**
 - retorna uma string com o caminho do arquivo ou diretório




Métodos *File*

- **String getParent()**
 - retorna uma string com o diretório-pai do arquivo ou diretório
- **long length()**
 - retorna o comprimento do arquivo em bytes (0 se for um diretório)


```
1.  import java.io.File;
3.  public class FileDemonstration{
4.      // apresenta informações sobre um arquivo/diretório específico
5.      public void analyzePath( String path ){
6.          // cria um objeto File baseado na entrada do usuário
7.          File name = new File( path );
9.          if ( name.exists() ) // {
10.             // apresenta informação do arquivo/diretório
11.             System.out.printf(
12.                 "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s",
13.                 name.getName(), " existe",
14.                 (name.isFile() ? "é um arquivo" : "não é um arquivo"),
15.                 (name.isDirectory() ? "é um diretório" :
16.                     "não é um diretório" ),
17.                 (name.isAbsolute() ? "é um caminho absoluto" :
18.                     "não é um caminho absoluto" ),
19.                 "Tamanho: ", name.length(),
20.                 "Caminho: ", name.getPath(),
21.                 "Absolute path: ", name.getAbsolutePath(),
22.                 "Pai: ", name.getParent() );
```

Métodos *File* - Exemplo



```
1.  if ( name.isDirectory() ) // imprime o conteúdo de um diretório
2.      {
3.          String directory[] = name.list();
4.          System.out.println( "\n\nConteúdo do diretório:\n" );
5.
6.          for ( String directoryName : directory )
7.              System.out.printf( "%s\n", directoryName );
8.      } // fim if
9.  } // fim if externo
10. else // imprime mensagem de erro
11.     {
12.         System.out.printf( "%s %s", path, "não existe." );
13.     } // fim else
14. } // fim metodo analyzePath
15. } // fim classe FileDemonstration
```



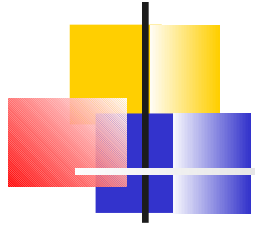
Métodos *File* - Exemplo

- Exercício: Escrever um programa que teste a classe **FileDemonstration**
 - o usuário pode fornecer uma string com o nome do arquivo/diretório para um objeto **scanner**
 - essa string é passada como argumento para o método **analyzePath** da classe **FileDemonstration**
 - (Resposta disponibilizada no fim do conjunto de slides)

Arquivos de texto de acesso seqüencial

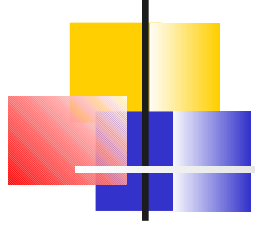


- Vistos como um fluxo de caracteres ou bytes
- Só podem ser percorridos do início para o fim (e não no sentido contrário)



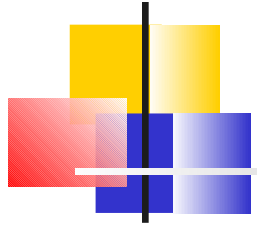
Leitura seqüencial

- Para trazer informação de uma origem (e.g., arquivo, memória etc), um programa Java abre um fluxo (*stream*) para leitura seqüencial.



Escrita seqüencial

- Para enviar informação a um destino (e.g., arquivo, memória etc) um programa Java abre um fluxo (*stream*) para escrita seqüencial.



Procedimento geral

Leitura

abrir fluxo

enquanto houver dados

ler

fechar fluxo

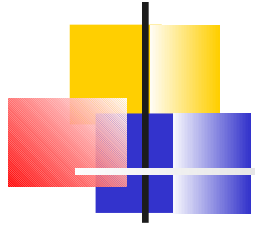
Gravação

abrir fluxo

enquanto houver dados

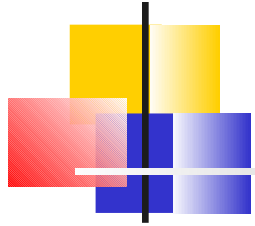
escrever

fechar fluxo



Classes de fluxos

- O procedimento para utilizar um fluxo de bytes ou um fluxo de caracteres é praticamente o mesmo
 - 1) criar um objeto de fluxo
 - 2) chamar seus métodos para enviar ou receber dados, dependendo se é um fluxo de entrada ou um fluxo de saída
 - 3) fechar o fluxo de dados



Fluxos de bytes

- Todos fluxos de bytes são uma subclasse de **InputStream** ou **OutputStream** (abstratas)
- Utilizados para manipulação de arquivos binários (ex: som, imagem ou dados em geral)
- Representam fluxos em arquivos que podem ser referenciados por um caminho na estrutura de diretórios e um nome de arquivo



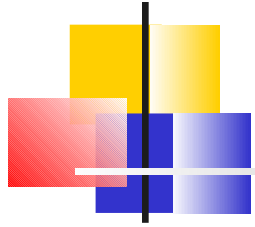
Fluxos de bytes

- Um fluxo de entrada de bytes pode ser criado com o construtor
 - **FileInputStream(String nome)**
- O argumento **nome** deverá ser o nome do arquivo a partir do qual os dados serão lidos
- É possível incluir no argumento o caminho onde se encontra o arquivo
 - permite que o arquivo esteja em uma pasta diferente daquela em que o aplicativo é executado



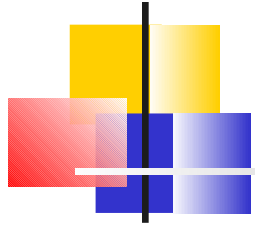
Fluxos de bytes

- Exemplo: a instrução a seguir cria um fluxo de entrada de bytes a partir do arquivo **score.dat**
- **`FileInputStream fluxo = new
FileInputStream("scores.dat")`**



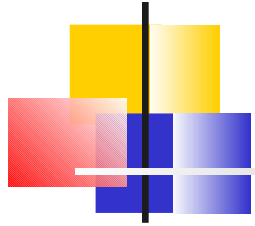
Fluxos de bytes

- Depois que um fluxo de entrada de bytes foi criado, é possível ler dados do fluxo, chamando seu método **read**
 - **int read()**: retorna o próximo byte no fluxo como um inteiro
 - **int read(byte[], int, int)**: lê bytes para o array de bytes especificado, com o ponto de partida indicado e número de bytes lidos



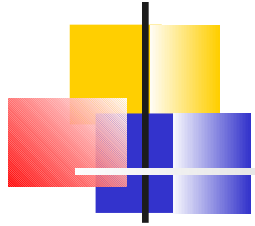
Fluxos de bytes

- Se o método **read** retornar **-1** significa que o final do arquivo foi alcançado
- Terminada a leitura dos dados o fluxo deve ser fechado chamando-se o seu método **close()**



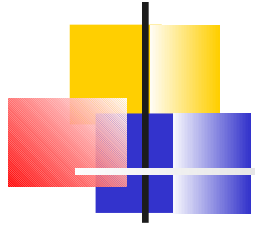
Fluxos de bytes

- Um fluxo de saída de bytes pode ser criado com o construtor
- **`FileOutputStream(String nome)`**
- É possível escrever dados do fluxo, chamando seu método **`write(int)`** ou **`write(byte[], int, int)`**
- Terminada a escrita dos dados o fluxo deve ser fechado chamando-se o seu método **`close()`**



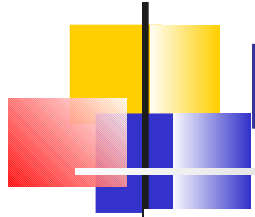
Fluxos de caracteres

- Usados para lidar com qualquer texto que seja representado pelo conjunto de caracteres Unicode de 16 bits
 - arquivos de texto puro
 - documentos HTML
 - arquivos fontes Java

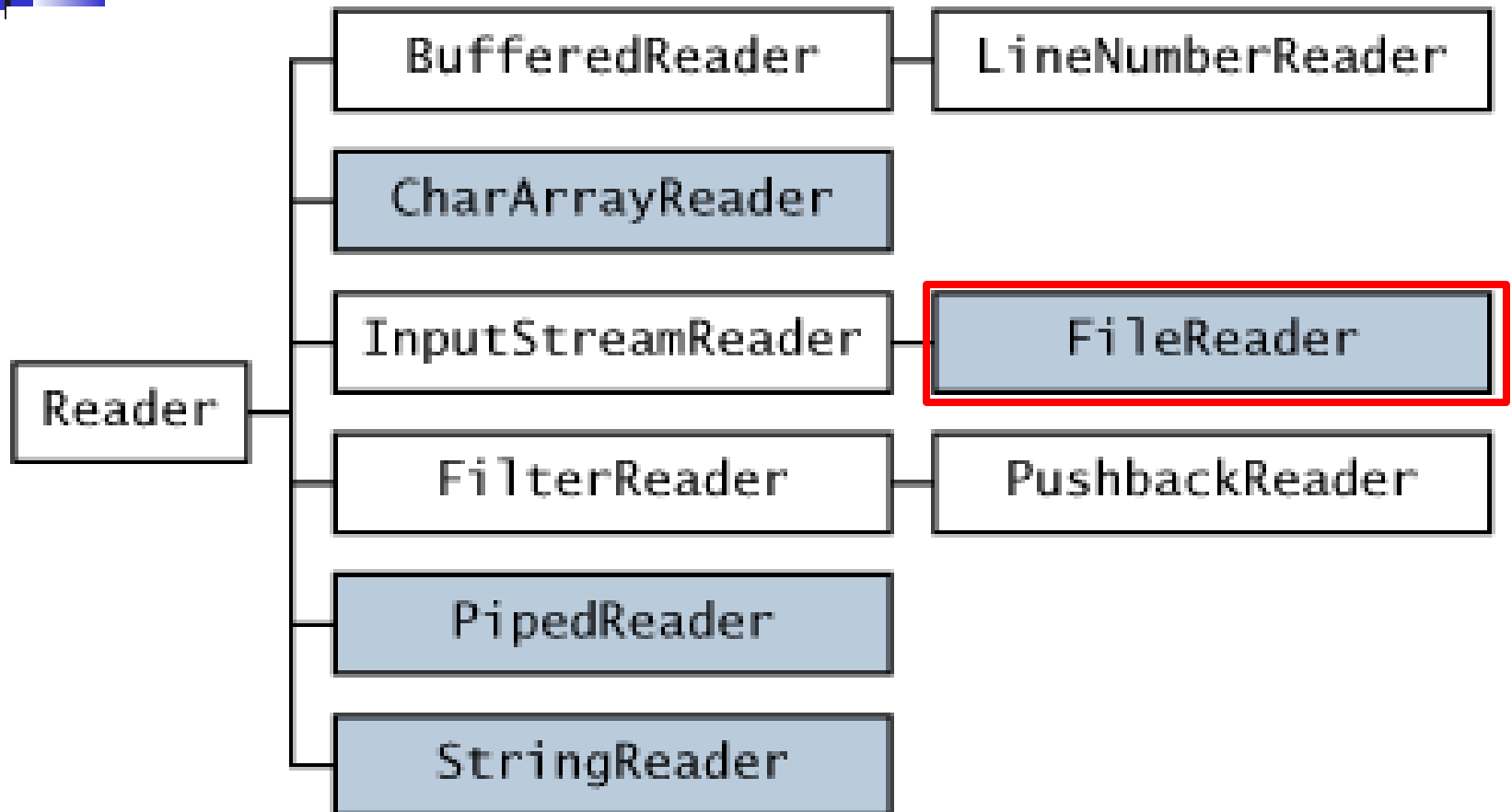


Fluxos de caracteres

- As classes usadas para ler e escrever fluxos de caracteres são todas derivadas das classes **Reader** e **Writer**
- **FileReader** é a classe principal usada para a leitura de fluxos de caracteres em um arquivo
 - subclasse de **InputStreamReader**, que lê um fluxo de bytes e converte os bytes para valores inteiros



Leitura de caracteres (16 bits)





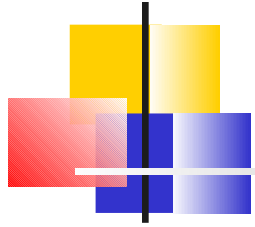
Fluxos de caracteres

- Um fluxo de entrada de caracteres é associado a um nome de arquivo usando o construtor
 - **FileReader (String nome)**
- Exemplo: a instrução a seguir cria um fluxo de entrada de caracteres e o associa a um arquivo texto
 - **FileReader fluxo = new FileReader (“index.txt”)**



Fluxos de caracteres

- Depois que um fluxo de entrada de caracteres foi criado, é possível ler dados do fluxo, chamando seu método **read**
 - **read()**: retorna o próximo caractere no fluxo como um inteiro
 - **read(char[], int, int)**: lê caracteres para o array de caracteres especificado, com o ponto de partida indicado e número de caracteres lidos



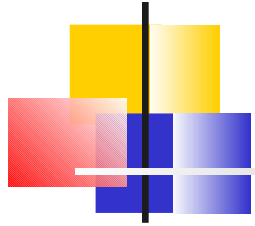
Fluxos de caracteres

- Como o método **read** retorna um inteiro, é preciso converter esse dado antes de ser
 - exibido
 - armazenado em um array
 - usado para formar uma string, etc.
- O inteiro retornado é um código numérico que representa o caractere no conjunto de caracteres Unicode

Exemplo

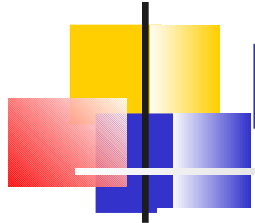
```
import java.io.*;
public class ReadCaracteres{
    public static void main(String[] args){
        int i;
        try{
            FileReader entrada = new FileReader("exemplo.txt");

            while (true){
                i = entrada.read();
                if ( i == -1 ) break;
                char c = (char) i;
                System.out.print( c );
            }
            System.out.println();
            entrada.close();
        }
        catch (IOException e){
            System.err.println(e);
        }
    }
}
```

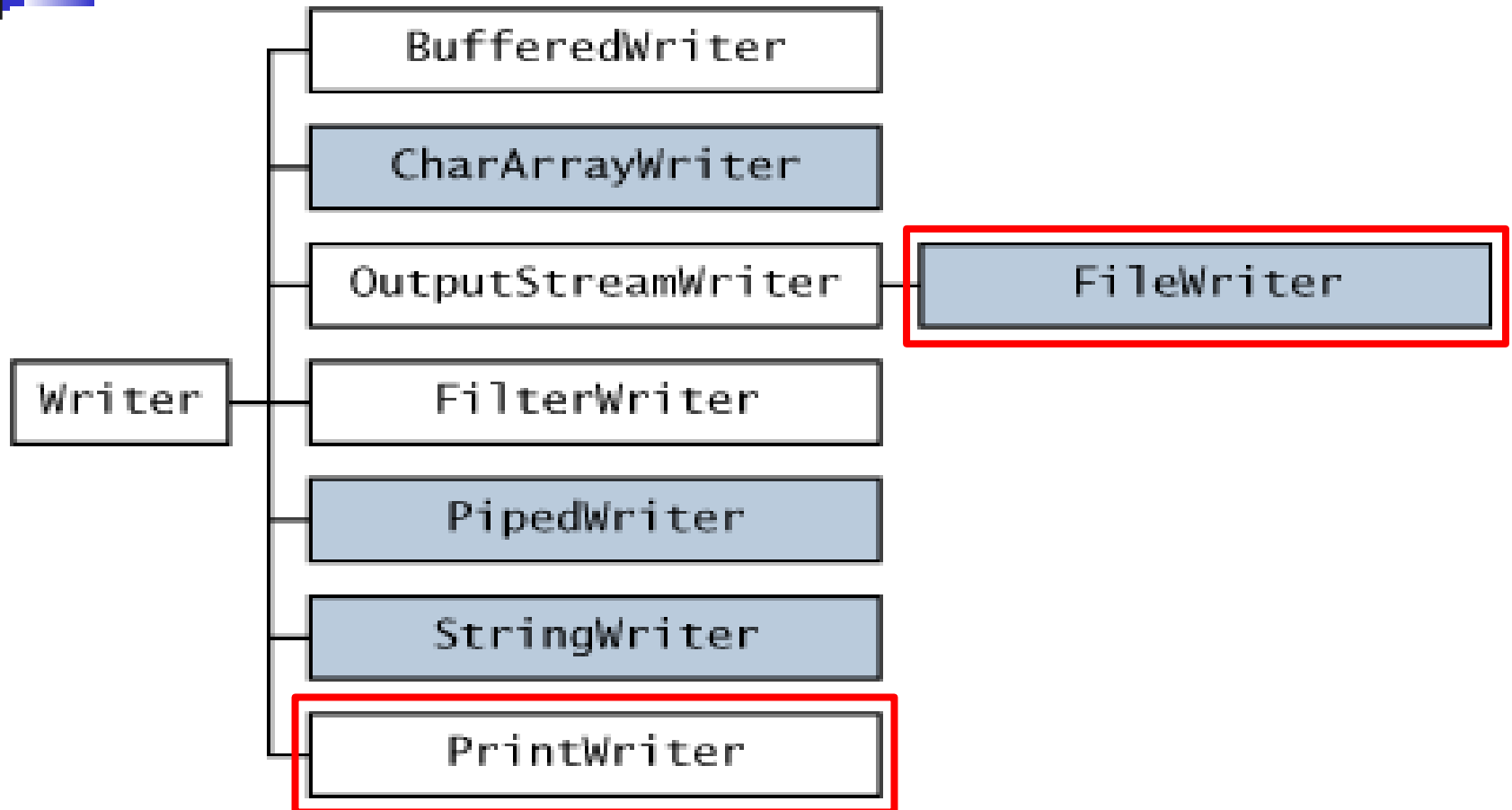


Fluxos de caracteres

- A classe **FileWriter** é a classe usada para gravar um fluxo de caracteres em um arquivo
 - subclasse de **OutputStreamReader**, que converte códigos de caractere Unicode em bytes



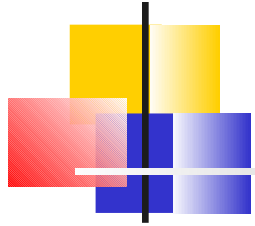
Escrita de caracteres (16 bits)





Fluxos de caracteres

- Existem dois construtores de **FileWriter**
 - **FileWriter (String nome)**
 - **FileWriter (String nome, boolean anexo)**
- O **nome** indica o nome do arquivo ao qual o fluxo de saída será direcionado (pode incluir o caminho)
- O argumento **anexo** será **true** se o fluxo tiver que ser anexado a um arquivo de texto existente



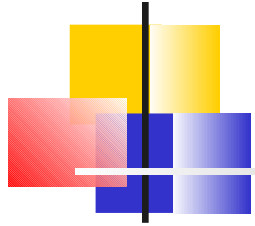
Fluxos de caracteres

- Três métodos de **FileWriter** podem ser usados para gravar dados em um fluxo:
 - **write(int):** grava um caractere
 - **write(char[], int, int):** grava caracteres do array de caracteres especificado, com o ponto de partida indicado e número de caracteres a serem gravados
 - **write(String, int, int):** grava caracteres da string especificada, com o ponto de partida indicado e número de caracteres a serem gravados



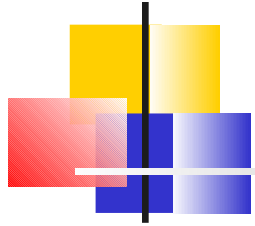
Exemplo: cópia de arquivo

```
public static void exemplo() throws IOException {  
    File arq_entrada = new File("entrada.txt");  
    File arq_saida = new File("saida.txt");  
  
    FileReader entrada = new FileReader(arq_entrada);  
    FileWriter saida = new FileWriter(arq_saida);  
  
    int c;  
    // -1 indica final de arquivo de caracteres  
    while ((c = entrada.read()) != -1)  
        saida.write(c);  
    entrada.close();  
    saida.close();  
}
```



Arquivos de dados

- Arquivos de texto não são convenientes para manipulação de dados em geral
- É possível utilizar fluxos de entrada e saída de dados das classes **DataInputStream** e **DataOutputStream**
- Esses fluxos filtram um fluxo de bytes existente de modo que tipos primitivos (*char*, *int*, *double* etc) possam ser lidos ou escritos



Abertura do fluxo

- Associando um arquivo

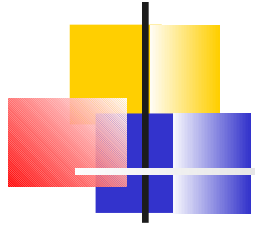
```
File arquivo = new File("dados.bin");
```

- Para leitura

```
DataInputStream entrada =  
    new DataInputStream(  
        new FileInputStream(arquivo));
```

- Para escrita

```
DataOutputStream saida =  
    new DataOutputStream(  
        new FileOutputStream(arquivo));
```



Leitura, escrita e fechamento

- Leitura (pode gerar *EOFException*)

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

- Escrita

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

- Fechamento

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



Exemplo: pedido de compra

- Considere a construção de um arquivo com dados (binários) em forma tabular:

<u>preço</u>	<u>quantidade</u>	<u>descrição</u>
10.00	12	mouse óptico
82.34	24	teclado
26.50	6	leitor cd-rom

- Os dados estão armazenados em arrays



Criação da tabela

```
File arquivo = new File("precos.bin");
DataOutputStream saida = new DataOutputStream(
    new
    FileOutputStream(arquivo));
for (int i = 0; i < precos.length; i ++) {
    saida.writeDouble(precos[i]);
    saida.writeChar('\t');
    saida.writeInt(quantidades[i]);
    saida.writeChar('\t');
    saida.writeChars(descricoes[i]);
    saida.writeChar('\n');
}
saida.close();
```



Leitura da tabela

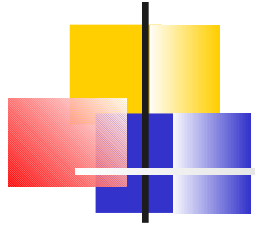
```
DataInputStream entrada =      new DataInputStream(  
    FileInputStream("precos.bin"));      new  
try {  
    while (true) {  
        preço = entrada.readDouble();  
        entrada.readChar();           // despreza o tab  
        quantidade = entrada.readInt();  
        entrada.readChar();           // despreza o tab  
        // etc...  
    }  
} catch (EOFException e) { // fim de arquivo }  
entrada.close();
```




Leitura da tabela

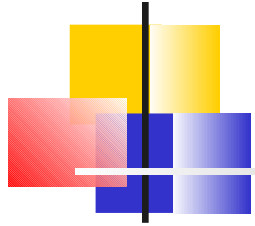
```
DataInputStream entrada = new DataInputStream(
    new FileInputStream("precos.bin"));
try {
    while (true) {
        preço = entrada.readDouble();
        entrada.readChar();          // despreza o tab
        quantidade = entrada.readInt();
        entrada.readChar();          // despreza o tab
        // etc
    }
} catch (EOFException e) { }
entrada.close();
```

Não existe **readChars**! Deve-se ler um caractere por vez em um loop



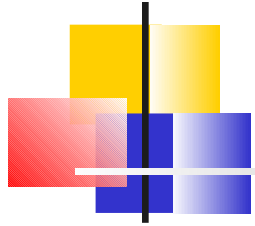
E/S de alto nível

- Serialização de objetos
 - Lê e escreve objetos inteiros em arquivo;
 - Arquivo em formato binário.



Arquivos de objetos

- Classe de dados deve implementar a interface **Serializable**
- A leitura (**ObjectInputStream**) e escrita (**ObjectOutputStream**) serial de objetos são filtros acoplados a um fluxo principal
- Permitem a leitura e escrita de objetos inteiros, incluindo suas referências a outros objetos
- Inclui suporte a tipos **Collection**



Arquivos de objetos

- Serialização de objetos: mecanismo para ler ou gravar um objeto inteiro a partir de um arquivo
 - realizada com fluxos de bytes
- Objeto serializado: representado como uma sequência de bytes que inclui:
 - os dados do objeto
 - as informações sobre o tipo do objeto
 - os tipos dos dados armazenados no objeto



Abertura do fluxo

- Associando um arquivo

```
File arquivo = new File("meusobjetos.bin");
```

- Para leitura

```
ObjectInputStream entrada =  
    new ObjectInputStream(  
        new FileInputStream(arquivo));
```

- Para escrita

```
ObjectOutputStream saida =  
    new ObjectOutputStream(  
        new FileOutputStream(arquivo));
```



Leitura, escrita e fechamento

- Leitura de dados (pode gerar *EOFException*)
`objeto = (Tipo) entrada.readObject();`
- Escrita de dados
`saida.writeObject(objeto);`
- Fechamento do arquivo
`entrada.close();`
`saida.close();`



Exemplo

- Considere uma classe representando registros de itens em um estoque (nome do produto, quantidade e valor);

```
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);
    } ...}
```



Exemplo

- Escrevendo em um fluxo de objetos

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




Exemplo

- Lendo a partir de um fluxo de objetos

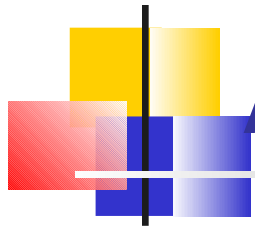
```
...  
Produto item1 = new Produto();  
try {  
    FileInputStream arq = new FileInputStream("tem.dat");  
    ObjectInputStream objarq = new ObjectInputStream(arq);  
    item1 = (Produto) objarq.readObject();  
    objarq.close();  
}  
catch(IOException e) {  
    System.out.println(e.getMessage());  
    e.printStackTrace();  
}  
...
```



Arquivos 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);
```

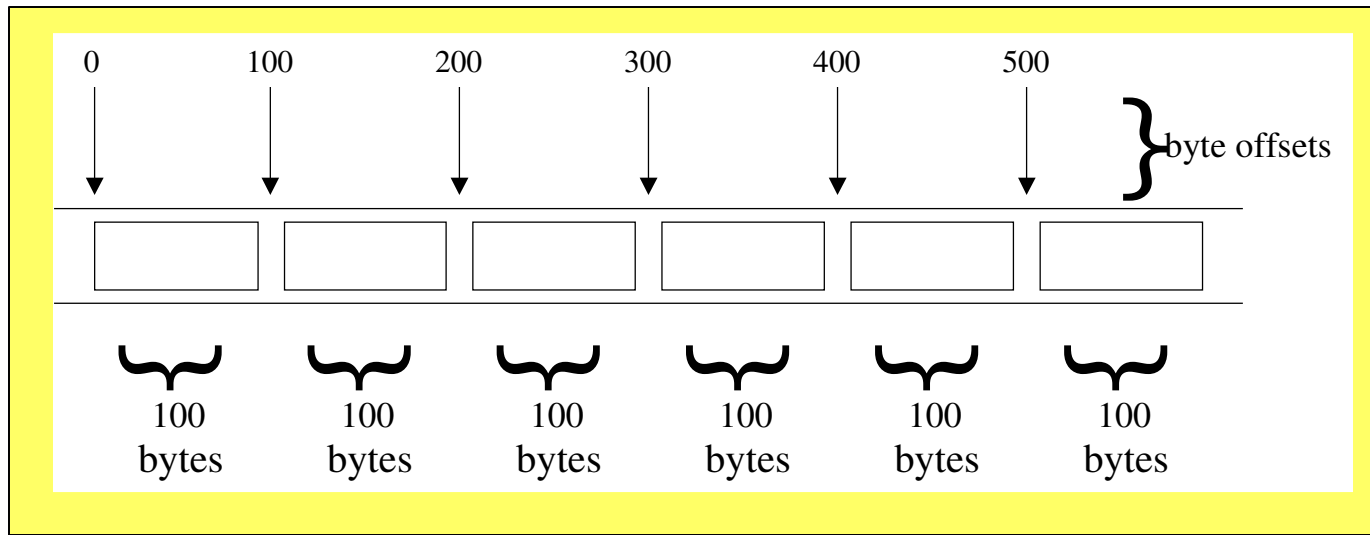


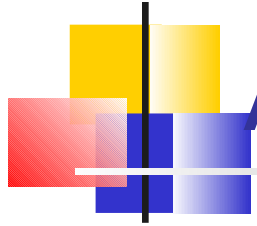
Arquivos de Acesso Aleatório

- 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**) ou “leitura e gravação” (**rw**)

Arquivos de Acesso Aleatório

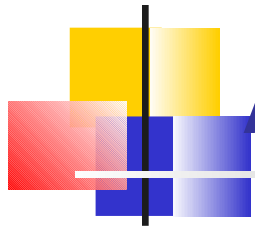
- Podem ser organizados de forma simples usando registros de tamanho fixo
 - Facilita o cálculo da localização exata de qualquer registro em relação ao início do arquivo.





Arquivos de Acesso Aleatório

- Os registros podem ser acessados diretamente, por meio de um ponteiro (ponteiro de arquivo)
- Inserção de dados não destrói outros registros
- Dados previamente armazenados podem ser atualizados sem sobrescrever outros



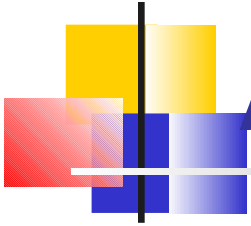
Arquivos de Acesso Aleatório

- 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(i.e., 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 = arq.getFilePointer();`
- Deslocamento para posição específica:
`arq.seek(posição);`
- Avanço de n posições:
`arq.skipBytes(n);`



Arquivos de Acesso Aleatório

- Abertura do arquivo

- Para leitura

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

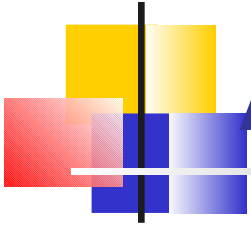
- Para escrita

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

- Para leitura e escrita

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





# Arquivos de Acesso Aleatório

---

- Leitura (pode gerar *EOFException*)

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

- Escrita

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

Strings devem ser  
construídas com  
**readChar**

- Fechamento

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

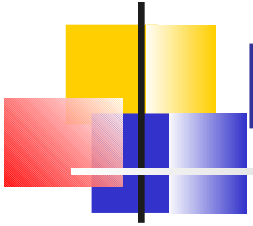
```
saida.close();
```



# 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
  - inclusão e atualização direta



# Exemplo: registros bancários

---

- A classe Registro

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



# Exemplo: registros bancários

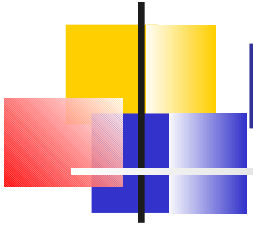
---

- Escrita da classe Registro

Array de objetos Registro

```
RandomAccessFile saida = new
RandomAccessFile("dados.bin", "rw") ;
for (int i = 0; i < clientes.length; i++) {
 saida.writeInt(i);
 saida.writeChars(clientes.nome[i]);
 saida.writeDouble(clientes.saldo[i]);
}
saida.close();
```

Uma alternativa melhor: definir um  
método de escrita para a classe Registro



# Exemplo: registros bancários

---

- Busca de um registro

```
// assume o arquivo saida já aberto
// cc é o nro. da conta a localizar

saida.seek((cc) * Registro.tamanho());

// a classe Registro implementa o método tamanho, que
// retorna uma constante inteira representando o nro.
// de bytes que ocupa
```



# 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 *Integer*, *Double* etc têm seu tamanho (em bits) dado por **SIZE**:

**TamanhoInteiro = Integer.SIZE / 8;**



# Exemplo: registros bancários

---

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

```
public int tamanho () {
 return (
 (Integer.SIZE/8) +
 15 * 2 + // string
 de 15 chars.
 (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.
- Prática comum: criar na própria classe Registro um método **escrever(arquivo)** para escrever um registro na posição atual do arquivo especificado





# Exemplo: registros bancários

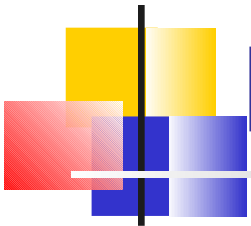
---

- Escrita de um registro:

```
public void escrever(RandomAccessFile f)
 throws IOException {
 f.writeInt(nroconta);
 StringBuffer b = new StringBuffer(nome);
 b.setLength(15);
 f.writeChars(b.toString()
 f.writeDouble(saldo);
}
```



Cria e manipula strings modificáveis



# 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();
}
```



# Resumo

---

- Foi apresentado os recursos da linguagem Java para tratamento de arquivos de diferentes formatos, a saber,
  - Texto; binário; binários com objetos serializados e com acesso aleatório.



# Referências

---

- H.M. Deitel & P. J. Deitel, Java – como programar, 6a. Edição, Pearson Prentice-Hall, São Paulo, 2005. Capítulo 14.



# Exercício

---

- Defina uma classe para registro de disciplinas, contendo informações sobre o nome da disciplina, seu código e sua carga horária
  - 1) Escreva métodos para leitura e escrita de objetos dessa classe usando arquivos de objetos
  - 2) Escreva métodos para leitura, escrita e modificação de objetos dessa classe utilizando E/S em arquivos de acesso aleatório
  - OBS: não esqueça de implementar o esquema de tratamento de exceções