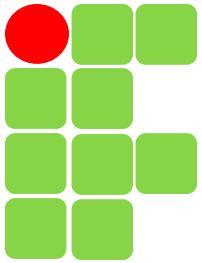
ADS – Análise e Desenvolvimento de Sistemas

INSTITUTO FEDERAL
SÃO PAULO
Campus São Carlos

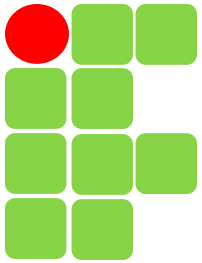
Linguagem de Programação3
Orientação a Objetos em Java
Aula 7

Profª. Maísa C. Duarte
maisa.duarte@ifsp.edu.br



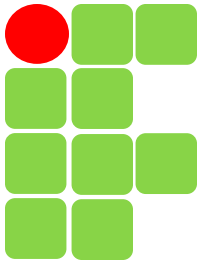
Agenda

- Threads;
- Exemplo de implementação;
- Exercícios;
- Leitura de Arquivo;
- Exercícios;

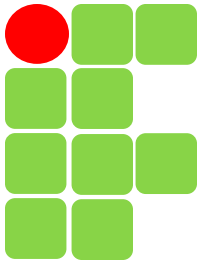


Introdução

- Thread é uma abstração que permite que uma aplicação execute mais de um trecho de código simultaneamente;
  - Ex: um método qualquer;

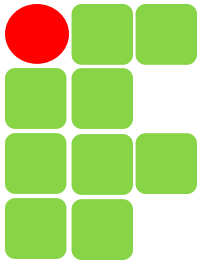


- Quando uma aplicação Java é executada:
  - A JVM cria um objeto do tipo Thread cuja tarefa a ser executada é descrita pelo método **main**().
  - A Thread é iniciada automaticamente.
  - Os comandos descritos pelo método **main**() são executados executados sequencialmente até que o método termine e a thread encerre.



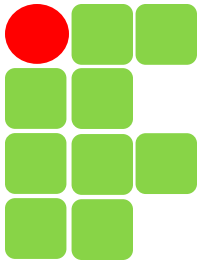
Introdução

- Threads, também chamados de processos leves, permitem múltiplas atividades dentro de um único processo;
- Um mesmo processo pode conter múltiplas threads que parecem executar ao mesmo tempo;
- Java é a primeira linguagem de programação a incluir o conceito de threads na própria linguagem.



Por que utilizar *Threads*?

- Há várias razões para utilizarmos threads. Entre elas:
  - Maior desempenho em ambientes multiprocessados;
  - Responsividade em interfaces gráficas;
  - Simplificação na modelagem de algumas aplicações.

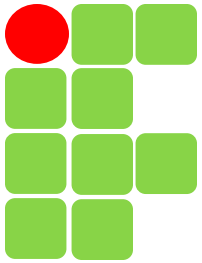


Criando *Threads*

- Há duas formas de criarmos uma thread em Java:

- Ou usamos herança e criamos uma classe que estende a classe **Thread**;

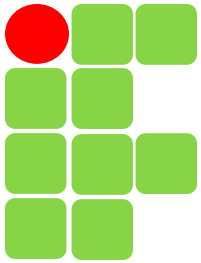
- Ou criamos uma classe que implementa a interface **Runnable**.



Usando Herança

- Usando **herança**, nossa classe deve sobrescrever o método **public void run()**:

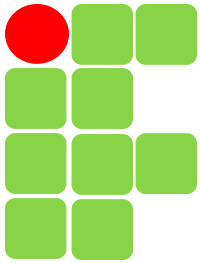
```
class Tarefa1 extends Thread {
    public void run() {
        for(int i=0; i<1000; i++)
            System.out.println("Usando herança");
    }
}
```



Usando a Interface *Runnable*

- A interface **Runnable** nos obriga a implementar o método **public void run()**:

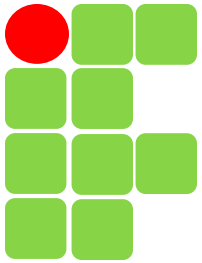
```
class Tarefa2 implements Runnable {
    public void run() {
        for(int i=0; i<1000; i++)
            System.out.println("Usando
              Runnable");
    }
}
```



Exemplo

- Para usar as classes **Tarefa1** e **Tarefa2** devemos fazer:

```
Thread threadComHeranca = new Tarefa1();
Thread threadComRunnable =
    new Thread(new Tarefa2());

threadComHeranca.start();
threadComRunnable.start();
```



Saída do Exemplo

- A saída do exemplo será ***mais ou menos*** a seguinte:

Usando Runnable

Usando Herança

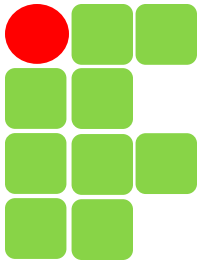
Usando Herança

Usando Runnable

Usando Herança

Usando Runnable

(...)

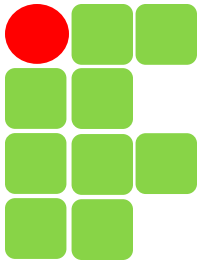


Principais métodos

- run(): é o método que executa as atividades de uma *thread*. Quando este método finaliza, a thread também termina.

- start(): método que dispara a execução de uma *thread*. Este método chama o método run( ) antes de terminar. Inicia a execução da thread (invocado somente uma vez);

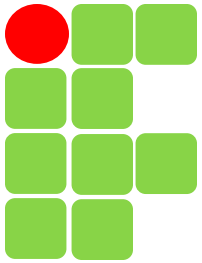
- sleep(int x): método que coloca a *thread* para dormir por **x** milissegundos.



Principais Métodos

- yield(): Faz com que a execução da thread corrente seja imediatamente suspensa, e outra thread seja escalonada;

- wait()/wait(t):Faz com que a thread fique suspensa até que seja explicitamente reativada por uma outra thread ou t acabe.

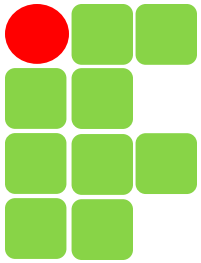


Principais métodos

- join( ): método que espera o término da *thread* para qual foi enviada a mensagem para ser liberada.

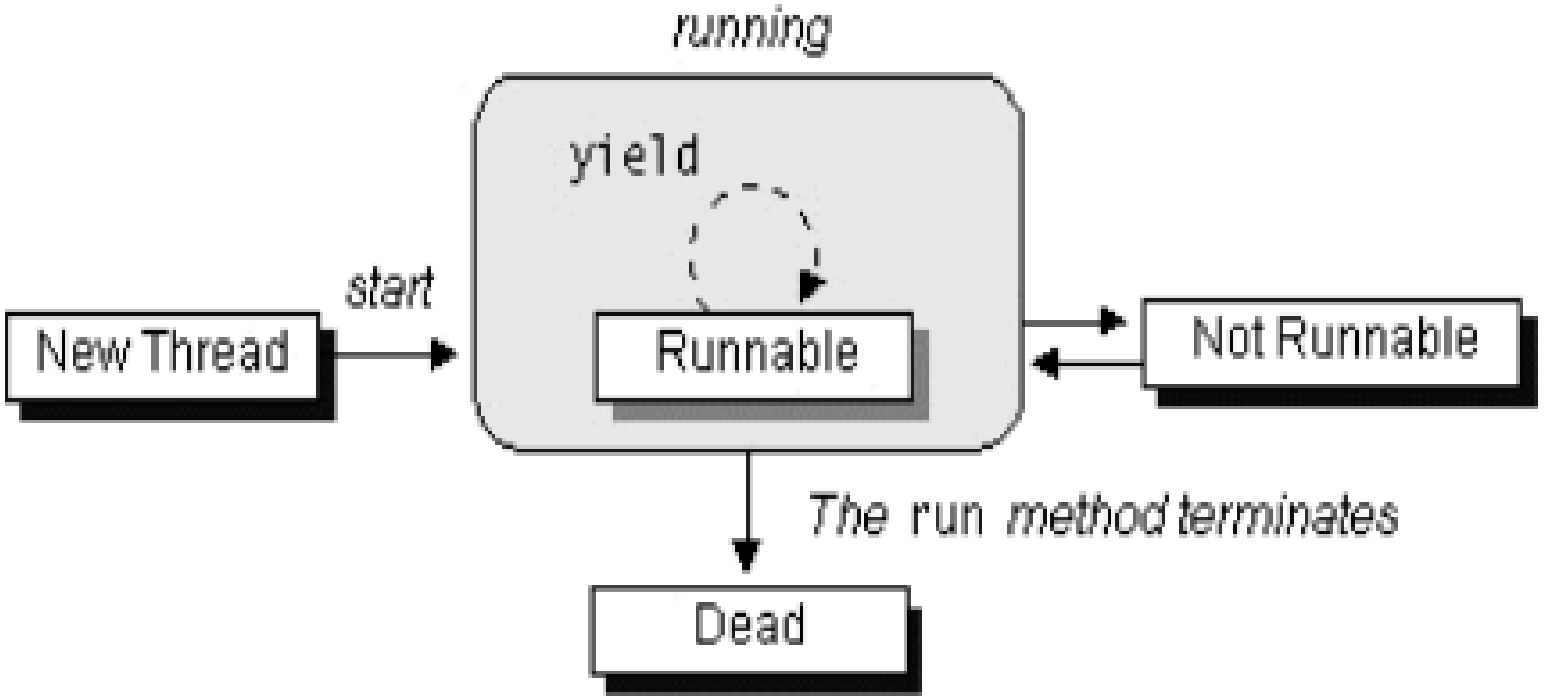
- interrupt( ): método que interrompe a execução de uma THREAD.

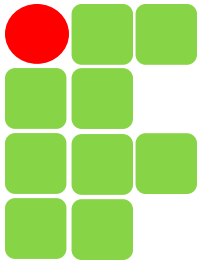
- interrupted( ): método que testa se uma *thread* está ou não interrompida.



Ciclo de vida de uma *thread*

Ciclo de Vida de um Thread



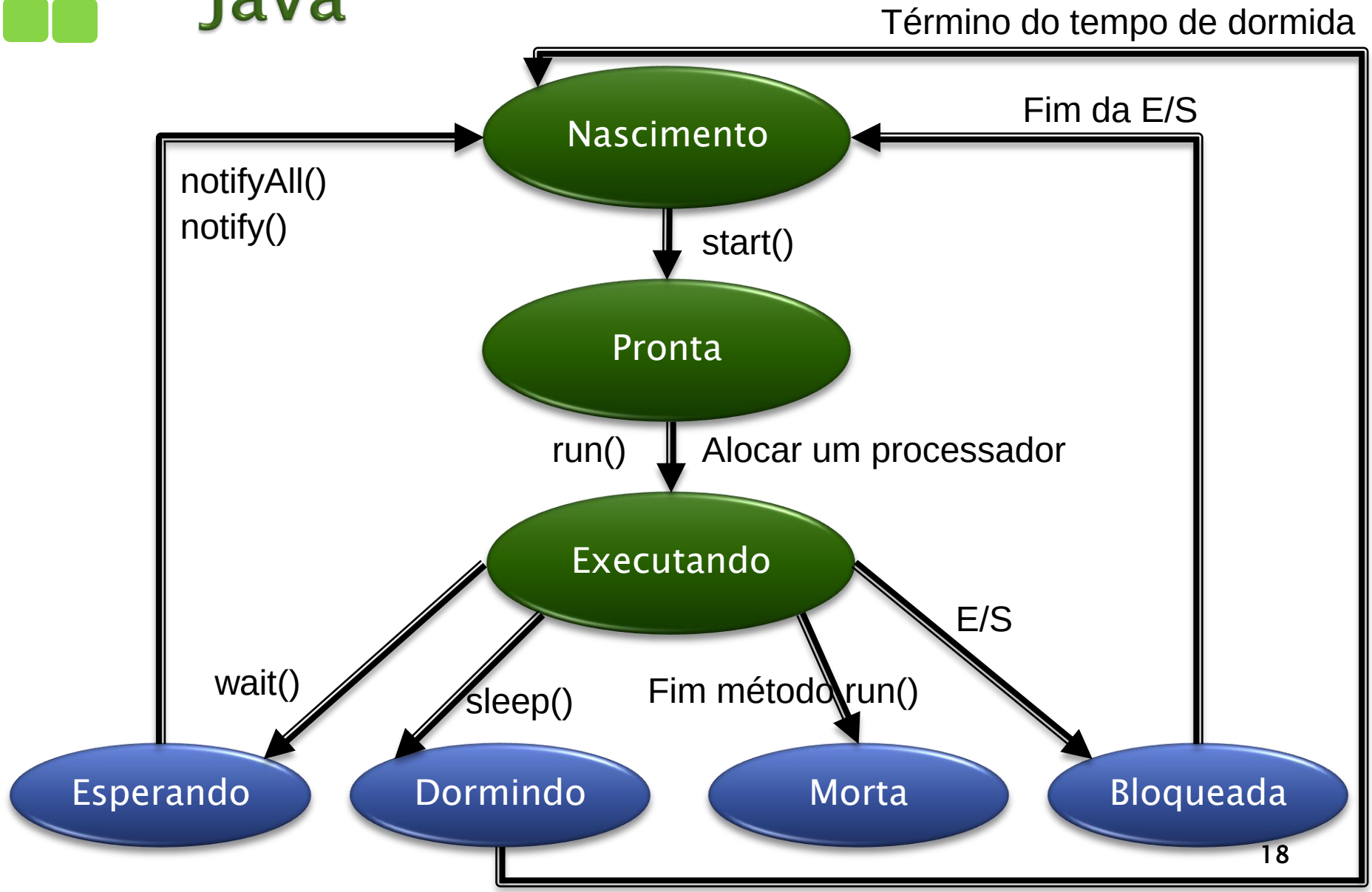


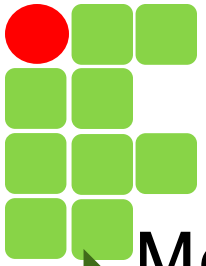
Estado Non Estado Non-Runnable

- Um thread pode entrar no estado non-runnable como resultado de:
  - a execução pela *thread* de uma chamada de I/O bloqueante;
  - invocação explícita dos métodos *sleep*() ou *wait*() do objeto *Thread*.

- O método yield() não torna o thread non-runnable, apenas transfere a execução para outra thread.

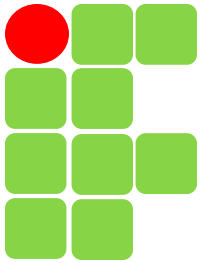
Estados de uma *thread* em Java





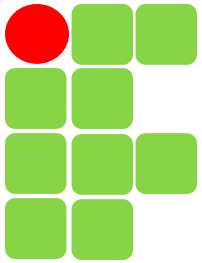
Escalonamento

- Mecanismo que determina como os threads irão utilizar tempo de CPU.
- Somente os threads no estado runnable são escalonados para ser executados.
- Java permite a atribuição de prioridades para as threads.
- Threads com menor prioridade são escalonados com menor frequência.
- Os threads são escalonados de forma preempetiva seguindo uma disciplina “round robin”.



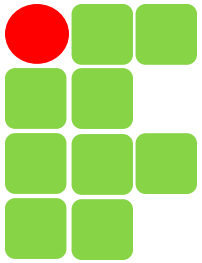
Round Robin

- Figura wikipedia...
- https://pt.wikipedia.org/wiki/Round-robin



Prioridades

- Todo thread possui uma prioridade que:
  - pode ser alterada com setPriority(int p)
  - pode ser lida com getPriority();
- Algumas constantes incluem:
  - Thread.MIN_PRIORITY
  - Thread.MAX_PRIORITY
  - Thread.NORM_PRIORITY (padrão);

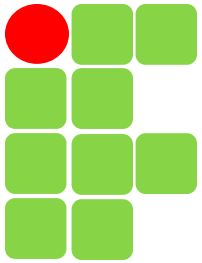


Prioridades

- A prioridade é determinada com um inteiro entre 1 e 10;

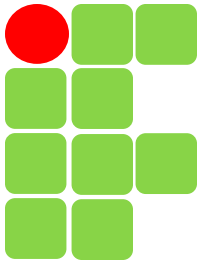
- A prioridade padrão é 5;
- Maior prioridade = 10;
- Menor prioridade = 1;

- A thread herda a prioridade da thread que a criou;



Sincronismo entre *Threads*

- O uso de memória compartilhada entre as *threads* obriga o programador a sincronizar as ações de suas threads.
- Para isso, Java provê *monitores* ou *locks*. Imagine um *lock* como uma permissão para que apenas uma *thread* possa utilizar um recurso por vez.
- Cada objeto em Java possui um *lock* e ele deve ser obtido através do comando *synchronized*.

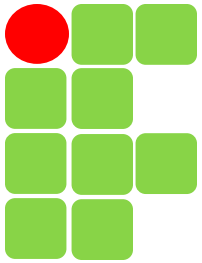


O comando *synchronized*

- Os seguintes usos do comando *synchronized* são equivalentes:

```
synchronized public void teste()
{
    façaAlgo();
}
```

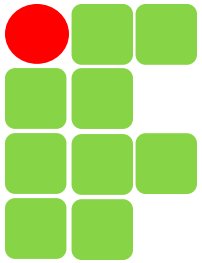
```
public void teste() {
    synchronized(this) {
        façaAlgo();
    }
}
```



wait(), notify() e notifyAll()

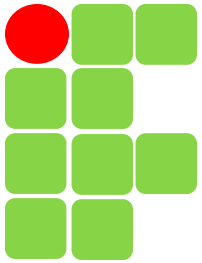
- São métodos da classe **Object** que implementam o conceito de monitores: notificam as threads indicando se estas devem ser suspensas ou se devem voltar a ficar em execução.

- O *lock* do objeto chamado pela *thread* para realizar as notificações será utilizado. Por isso, antes de chamar um dos três métodos, o *lock* deve ser obtido utilizando-se o comando *synchronized*.



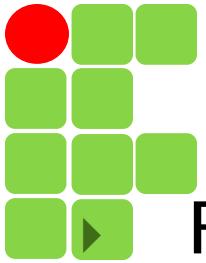
wait(), notify() e notifyAll()

- **wait()** - suspende a *thread* que chamou o método até que outra *thread* a acorde ou até que o tempo especificado como argumento tenha passado;
- **notify()** - acorda, se existir, alguma *thread* que esteja esperando um evento neste objeto;
- **notifyAll()** - acorda todas as *threads* que estejam esperando neste objeto.



Exercício 1 – Enviar Tarefa

- Refaçam o exemplo do Produtor Consumidor;
  - Faça comentários em cada bloco/linha (de acordo com o necessário);

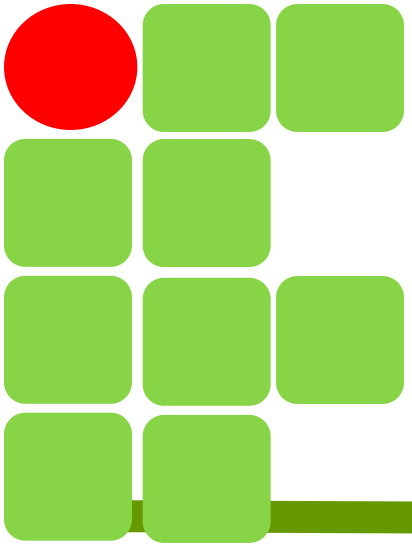


Exercício 2 – Enviar Tarefa

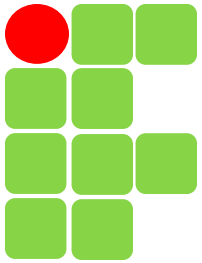
Façam um cronômetro utilizando threads: Crie a interface gráfica (como no exemplo mesmo);

- Classe Cronometro: Possui o método nextTick() para controle de horas, minutos, segundos e milissegundos (todos são atributos).
- Classe TemporizadorInterno: Tem Cronometro; É uma thread com sleep(1), após acordar a thread invoca o nextTick() do Cronometro;
- Classe TemporizadorExterno: Tem Cronometro e a Classe GUI; É uma thread com sleep(1000), após acordar atualiza a GUI.
- *Dica: na GUI crie métodos para alterar as labels, e invoque-os em Temporizador Externo;*

INSTITUTO FEDERAL
SÃO PAULO
Campus São Carlos

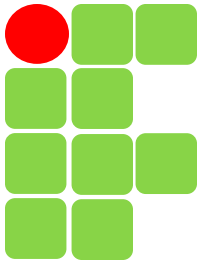


Manipulação de Arquivos



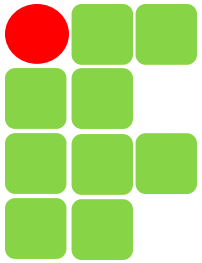
Introdução

- Arquivos fornecem aos sistemas uma alternativa para persistência de dados;
- Com isso, informações gravadas num momento podem ser recuperadas posteriormente;
- Em Java, interfaces e classes para a manipulação de arquivos estão disponíveis no pacote “java.io”;



Tipos de Arquivos

- Arquivos podem armazenar caracteres (arquivo texto) ou bytes (arquivos binários);
- Na prática, arquivos textos são abertos por editores de texto simples, enquanto que arquivos binários precisam ser abertos por programas específicos;
- Exemplos de arquivos:
  - Texto (qualquer arquivo que só armazene texto): .txt, .xml, .bat …
  - Binário (possuem uma codificação de bytes específica): .doc, .exe, .zip, …

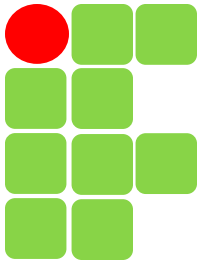


Manipulação de Arquivo: Classes principais

- **File**: “Representação abstrata de nome de caminhos de arquivos e diretórios”; Trabalha com caminhos, **não** é usada propriamente para ler ou escrever arquivos;
- **FileReader**: Usada para ler arquivos de caracteres.Seus métodos read() são de nível alto. Permite leitura de caracteres isolados, todo o stream de caracteres ou um número fixo de caracteres;
- **BufferedReader**: Usada para tornar classes Reader de nível mais baixo, como FileReader, mais eficientes e fáceis de usar;

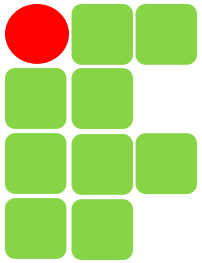
Buffer

- Local da memória física utilizado para armazenar temporariamente dados que estão sendo manipulados;



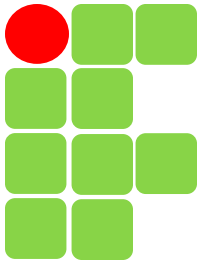
Porque BufferedReader é mais eficiente que FileReader?

- Um **BufferedReader** lê pedaços relativamente grandes de dados retirados do arquivo de uma só vez, e mantêm esses dados em um buffer.
- Quando solicita-se um caracter ou a próxima linha, esta é retirada do buffer;
- Possui métodos mais convenientes, como o readLine(), que obtém a próxima linha;



Manipulação de Arquivo: Classes principais

- **FileWriter**: usada para escrever arquivos em caracteres. Seus métodos write() permitem escrever caracteres ou strings em um arquivo;
- **BufferedWriter**: Usadas para tornar as classes de baixo nível, como FileWriters mais eficientes e fáceis de usar.



Porque BufferedWriter é mais eficiente que FileWriter?

- **Métodos BufferedWriter** escrevem pedaços relativamente grandes de dados no arquivo de uma só vez, minizando o número de operações para manipulação de arquivos;

- Fornece método newLine() para criação automática de separadores de linhas específicos à plataforma

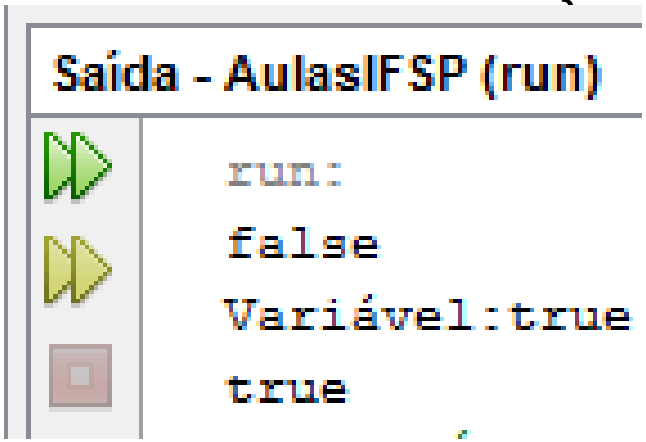
Criando arquivos com a classe File

- Objetos do tipo File são usados para representar os arquivos, mas não os dados ou diretórios que existem no disco físico;

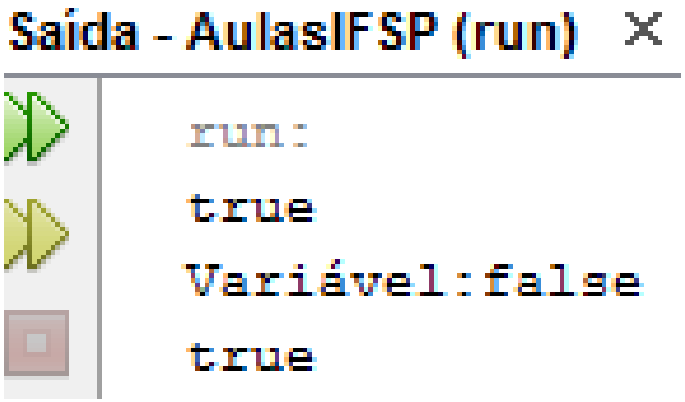
```
import java.io.*;
public class ManipulacaoFile {
    public static void main(String args[]){
        try {
            //Variável auxiliar
            boolean ConfirmaCriacao = false;
            //Objeto
            File arquivo = new File("Arquivo1.txt");
            //Procura por arquivo
            System.out.println(arquivo.exists());
            //Talvez crie o arquivo!
            ConfirmaCriacao = arquivo.createNewFile();
            //Procura por arquivo
            System.out.println("Variável:" + ConfirmaCriacao);
            System.out.println(arquivo.exists());
        } catch (IOException ex) {
           System.err.println("ERRO - FILE!!!");
        }
    }
}
```

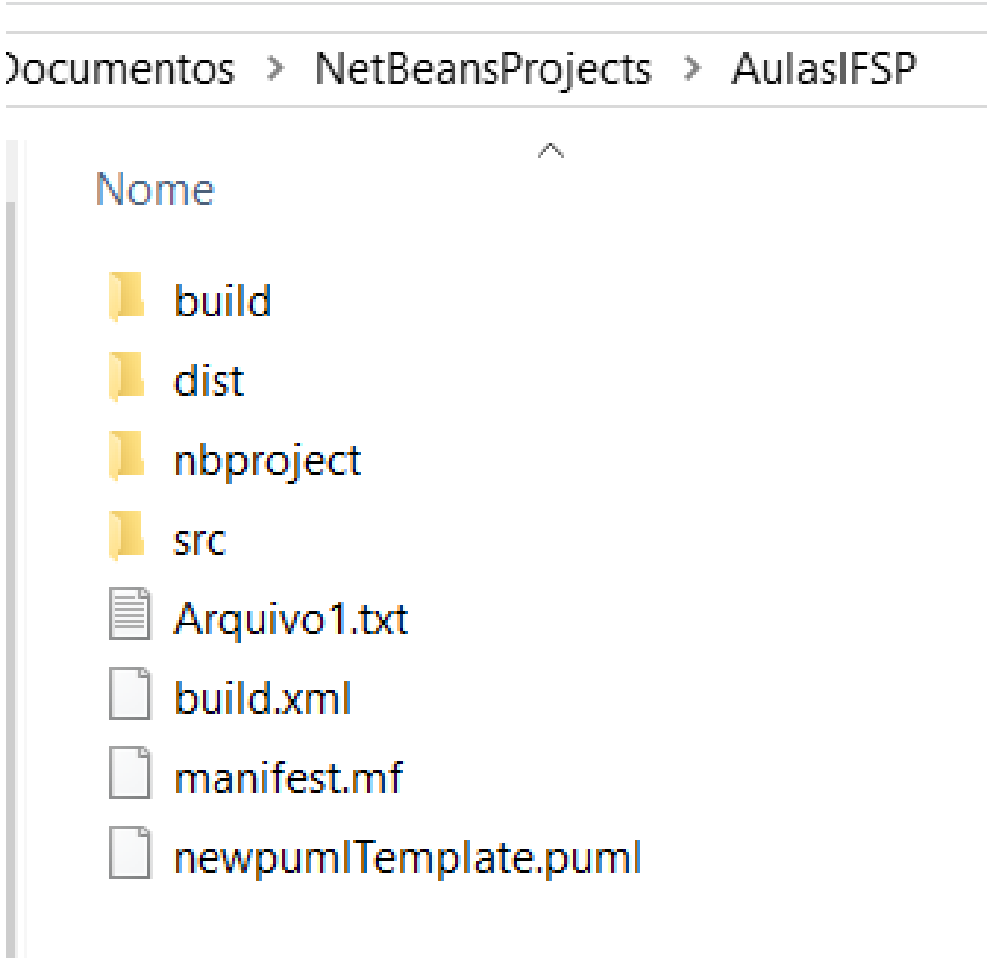
Criando arquivos com a classe File

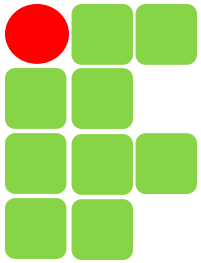
- Primeira execução:



- Segunda execução



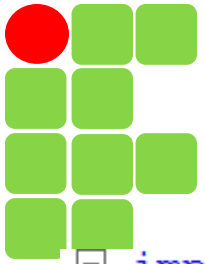




Usando FileWriter e FileReader

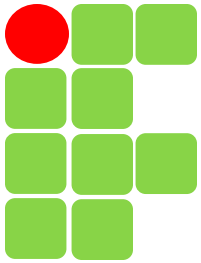
```java
import java.io.*;
public class ManipulacaoFileWriterFileReader {
    public static void main(String[] args) throws IOException {
        char[] input = new char[50];
        int size = 0;
        //Apenas objeto
        File arquivo = new File("Arquivo2.txt");
        //Cria o arquivo
        FileWriter fw = new FileWriter(arquivo);
        //Escreve caracteres com quebra de linha
        fw.write("Primeira Linha\r\n");
        fw.write("Segunda Linha\r\n");
        //Limpa antes de fechar
        fw.flush();
        //Fecha o arquivo
        fw.close();

        //Direciona o arquivo a ser lido
        FileReader fr = new FileReader(arquivo);
        //Nr. de bytes lidos
        size = fr.read(input);
        System.out.println("Bytes: " + size);
        for (char aux : input) {
            System.out.print(aux);
        }
        fr.close();
    }
}
```

```
Saída - AulasIFSP (run)
run:
31
Primeira Linha
Segunda Linha
```



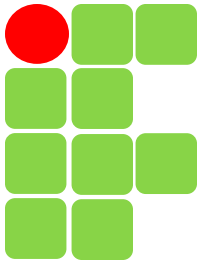
Usando o BufferedReader

```
import java.io.*;

public class ManipulacaoBufferedReader {

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

        String linha;
        //Mapeia o arquivo
        File arquivo = new File("Arquivo2.txt");
        //Obtem dados a partir de File/Arquivo
        FileReader fr = new FileReader(arquivo);
        //Obtem os dados a partir do reader
        //e armzena no buffer de leitura
        BufferedReader br = new BufferedReader(fr);
        while ((linha = br.readLine()) != null) {
            System.out.println(linha);
        }
    }
}
```



Usando o BufferedWriter

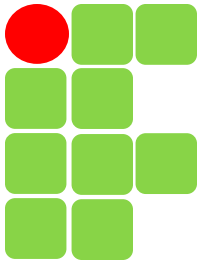
```
import java.util.Scanner;
public class ManipulacaoBufferedWriter {

    public static void main(String[] args) throws IOException {
        BufferedWriter buffWrite = new BufferedWriter(
                new FileWriter("Arquivo3.txt"));
        String linha = "";
        Scanner in = new Scanner(System.in);

        System.out.println("Escreva algo: ");
        linha = in.nextLine();

        buffWrite.append(linha + "\n");
        buffWrite.close();
    }
}
```



Abertura de Arquivos

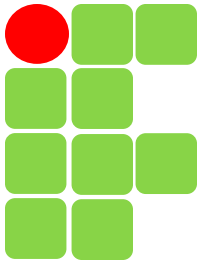
- Arquivos podem ser abertos para leitura (arquivo já existente) ou escrita (tanto faz);

- Esta abertura é usualmente realizada através da chamada à um construtor de classe apropriado *(new):*
  - *FileInputStream* e *FileOutputStream* para leitura e escrita de arquivos vistos como uma sequência de bytes;
  - *FileReader* e *FileWriter* para leitura e escrita de arquivos vistos como uma sequência de caracteres;



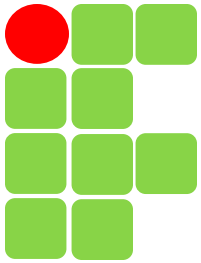
Fechamento de Arquivos

- Para o fechamento utilizamos o método *close();*
- Antes da chamada a este método, o arquivo manipulado fica "bloqueado" para o programa que o utiliza;
- Por isso, é uma prática recomendável fechar o arquivo imediatamente após a sua utilização.



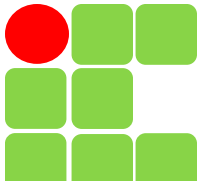
Manipulação de Arquivo txt

```
public void testArquivo() {
    String path = "C:\\saidas\\saidaa.txt";
    File arquivo = new File(path);

    try {
        if (!arquivo.exists()) {
            //cria um arquivo (vazio)
            arquivo.createNewFile();
        }
    } catch (IOException ex) {
        ex.printStackTrace();
    }
```



Manipulação de Arquivo txt

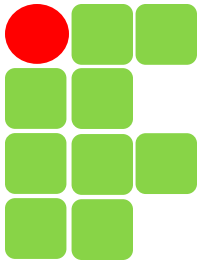
```
//listar seus arquivos e diretórios
File[] arquivos = arquivo.listFiles();
```



Manipulação de Arquivo txt

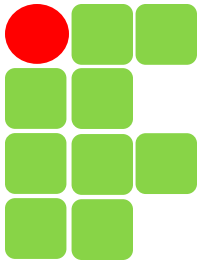
```
//escreve no arquivo e escreve no final
//se passar false por parâmetro ou não passar nada,
//o arquivo será limpo escrito do início novamente
FileWriter fw = new FileWriter(arquivo, true);
//Escrita direta
fw.write("Aqui é um texto de teste \n");

//Escrita utilizado a classe BufferedWriter
//Propicia métodos independentes do SO.
BufferedWriter bw = new BufferedWriter(fw);
bw.write("Texto a ser escrito no txt");
bw.newLine();
```

```
bw.close();
fw.close();
```



Manipulação de Arquivo txt

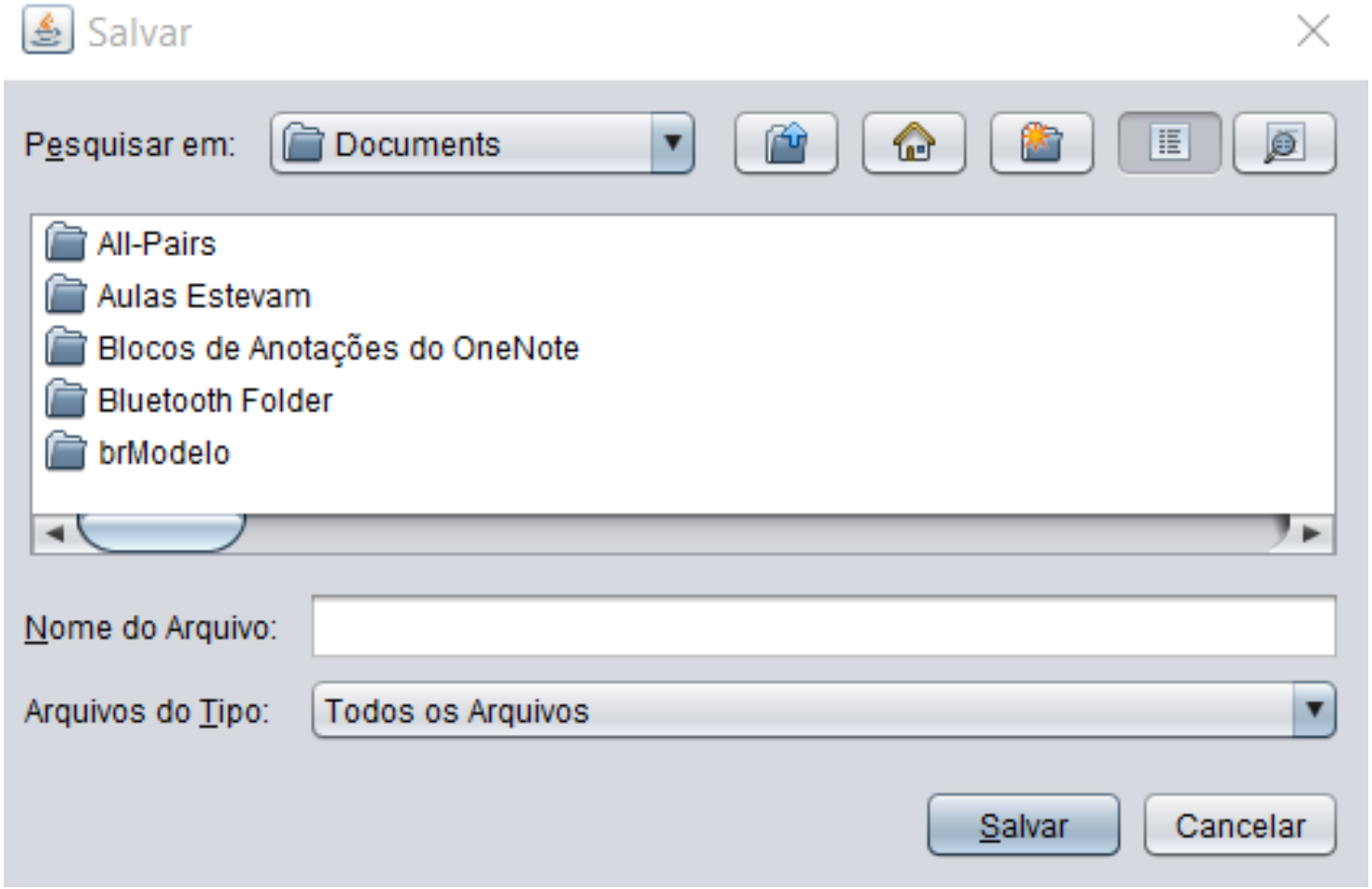
```
//leitura do arquivo
FileReader fr = new FileReader(arquivo);

BufferedReader br = new BufferedReader(fr);

//equanto houver mais linhas
while (br.ready()) {
    //lê a proxima linha
    String linha = br.readLine();

    //faz algo com a linha
    System.out.println(linha);
}

br.close();
fr.close();
```

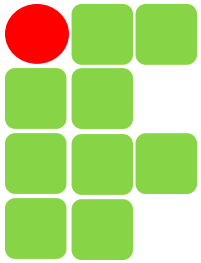


Exercício

- Usando BufferedReader e BufferedWriter, faça um programa que escreva um texto de acordo com a entrada do usuário e em seguida apresente o texto todo;
- ps: Não é necessária a interface gráfica

Componente JFileChooser





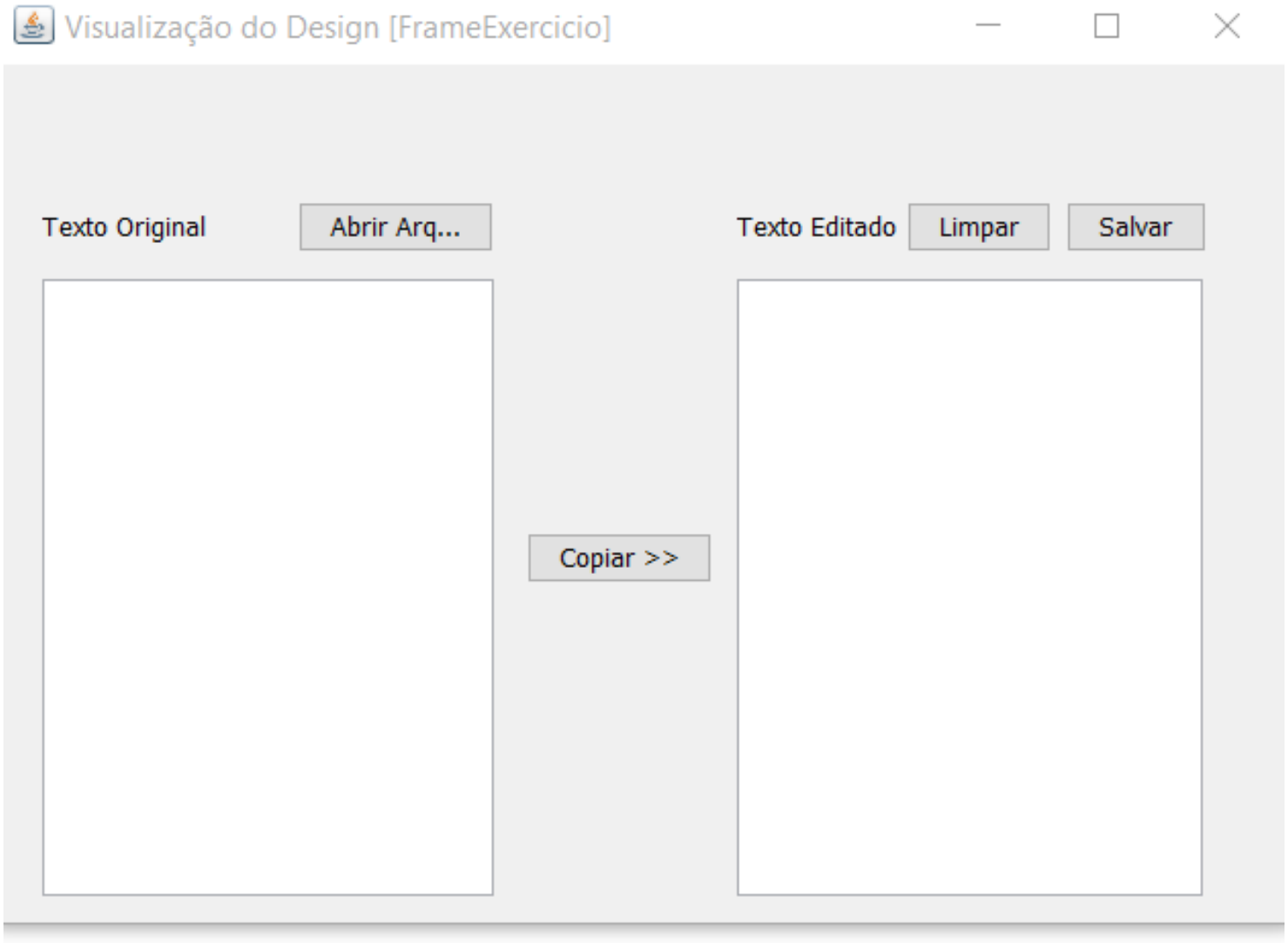
Componente JFileChooser

```
private void jButtonOpenFileActionPerformed(java.awt.event.ActionEven
    JFileChooser filePath = new JFileChooser();
    filePath.setFileSelectionMode(JFileChooser.FILES_ONLY);
    int i = filePath.showSaveDialog(null);
    if (i == 1) {
        jTextFieldTEXTO.setText("");
    } else {
        File arquivo = filePath.getSelectedFile();
        jTextFieldTEXTO.setText(arquivo.getPath());
    }
}
```

Exercício – Tarefa moodle

- Crie um programa com GUI inspirado no slide a seguir

Exercício


Visualização do Design [FrameExercicio]
Texto Original
Abrir Arq...
Texto Editado
Limpar
Salvar
Copiar >>