

Curso

Pacotes, Lambdas, Streams, Interfaces Gráficas

Atualizado até o Java 21 & Eclipse 2023-09



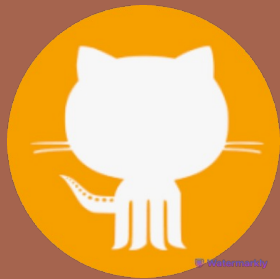
Prof. Msc. Antonio B. C. Sampaio Jr
ENGENHEIRO DE SOFTWARE & PROFESSOR

@abctreinamentos
@amazoncodebr

www.abctreinamentos.com.br
www.amazoncode.com.br



REPOSITÓRIO GITHUB



antonio-sampaio-jr / **pacotes-lambdas-streams**

CONTEÚDO PROGRAMÁTICO



- UNIDADE 1 – PACOTES, ERROS E EXCEÇÕES
- UNIDADE 2 – ANOTAÇÕES E ENTRADA/SAÍDA
 - Remoção Completa Applets Java e JWS [NOVO]
 - Gráficos, Fontes/Textos e Cores
 - Anotações
 - Streams de Entrada e Saída
 - Novos Métodos para Leitura e Escrita em Arquivos [NOVO]

CONTEÚDO PROGRAMÁTICO



- UNIDADE 2 – ANOTAÇÕES E ENTRADA/SAÍDA (Continuação)
 - Entrada e Saída de Dados com as classes Scanner e Formatter



UNIDADE 2

ANOTAÇÕES E ENTRADA/SAÍDA



Remoção Completa Applets Java & Java Web Start

Applets Java

- Applets Java são pequenas aplicações Java que podem ser executadas em um navegador web. Elas foram uma das primeiras formas de interatividade dinâmica na web, antes da popularização de tecnologias como Javascript.

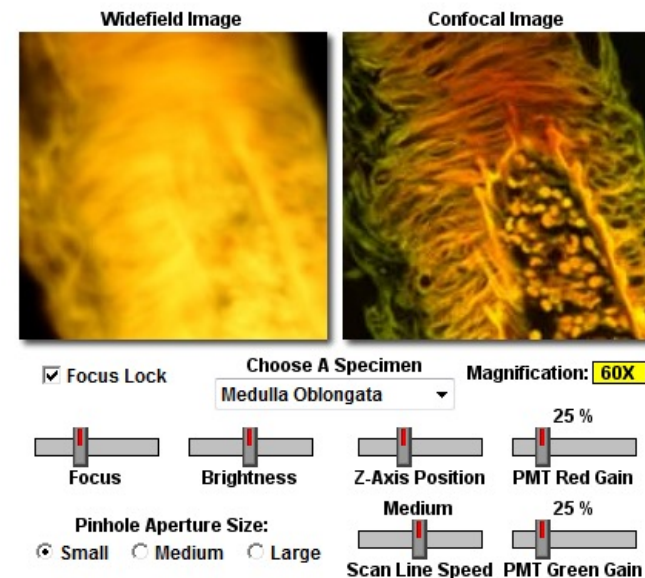
Laser Scanning Confocal Microscopy

Laser scanning confocal microscopes employ a pair of pinhole apertures to limit the specimen focal plane to a confined volume approximately a micron in size. Relatively thick specimens can be imaged in successive volumes by acquiring a series of sections along the optical (z) axis of the microscope.



Laser Scanning Confocal Microscopy

Laser scanning confocal microscopes employ a pair of pinhole apertures to limit the specimen focal plane to a confined volume approximately a micron in size. Relatively thick specimens can be imaged in successive volumes by acquiring a series of sections along the optical (z) axis of the microscope.



Applets Java

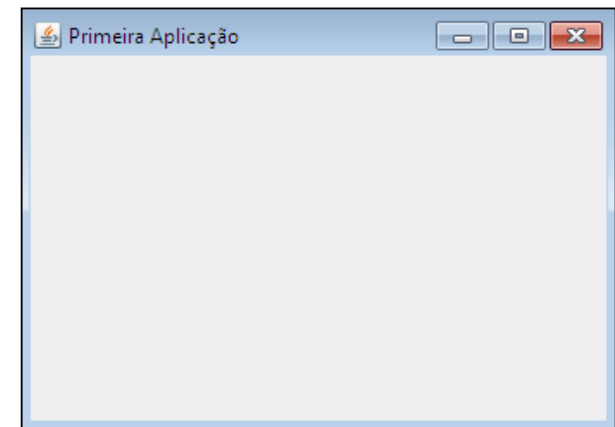
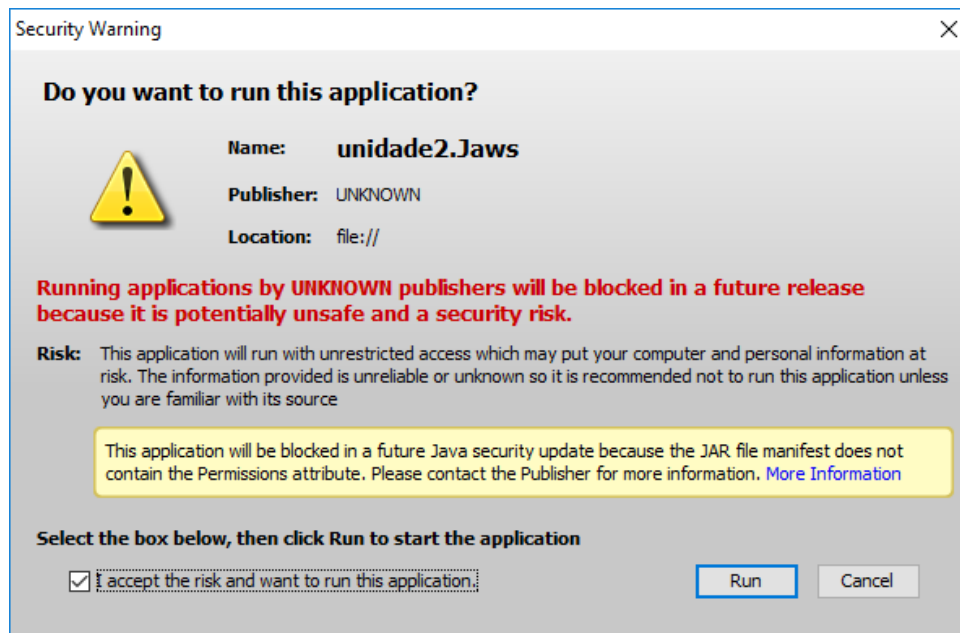
- Os applets eram escritos em Java e incorporados em páginas web usando a tag <applet>. Eles podiam realizar uma variedade de tarefas, desde jogos simples até aplicações mais complexas, como visualizadores de gráficos.
- Entretanto, devido a preocupações com segurança e a evolução das tecnologias web, o suporte a applets foi gradualmente descontinuado.
- O Java Applets foi descontinuado e removido do JDK a partir da versão Java 11, que foi lançada em setembro de 2018.
- Atualmente as aplicações web interativas são construídas principalmente usando tecnologias web como HTML, CSS e JavaScript, junto com frameworks e bibliotecas como React, Angular e Vue.js. Essas tecnologias oferecem uma experiência mais dinâmica e segura para os usuários.

Java Web Start

- Uma alternativa aos applets foi a tecnologia **Java Web Start (JAWS)** que possibilitava a execução de uma aplicação Java via rede (local ou Internet), sem a necessidade de um browser.
- Essa tecnologia permitia aos desenvolvedores distribuir e executar aplicações Java diretamente a partir de um navegador web. Ele foi projetado para facilitar a implantação de aplicações Java em ambientes corporativos e para simplificar a experiência do usuário final.
- A partir da versão Java 9, o Java Web Start foi descontinuado e removido da distribuição padrão do JDK.
- Desde o Java 9 essa tecnologia já estava 'marcada' como *deprecated*. A partir do Java 11, ela foi removida completamente da API do Java.

Exemplo do JAWS

- Dar um duplo clique no arquivo **jaws.jnlp**.





Gráficos, Fontes/Textos e Cores

Gráficos, Fontes/Textos e Cores

- Para desenhar no Java é necessário um contexto gráfico. Um objeto da classe **Graphics** controla o modo como a informação é desenhada, pois contém métodos para desenhar, manipular fontes e cores.
- Um objeto da classe **Graphics** é passado pelo sistema ao método **paint(...)** como argumento, quando uma operação **paint** ocorre.
- Quando um applet é inicialmente executado, o método **paint(...)** é automaticamente chamado (depois da chamada dos métodos **init()** e **start()**).
- Para que **paint(...)** volte novamente a ser chamado é necessário que ocorra um evento tal como um redimensionamento ("*resizing*") do applet.
- Se o programador necessitar chamar um método que efetue o *painting*, deve chamar o método **repaint()** (**public void repaint()**) que efetua um "*clear*" seguido de um "*paint*".

java.awt.Graphics

- Representa o contexto gráfico de cada componente de um applet.
- A classe **Graphics** possui um conjunto de métodos para desenhar: 1. Linhas, 2. Retângulos, 3. Retângulos arredondados, 4. Polígonos, 5. Ovais e 6. Arcos.

1. Desenho de Linhas

public void drawLine(int x1, int y1, int x2, int y2)

⇒ desenha uma linha entre os pontos [x1, y1] e [x2, y2].

2. Desenho de Retângulos

public void drawRect(int x, int y, int width, int heigh)

public void fillRect(int x, int y, int width, int heigh)

public void clearRect(int x, int y, int width, int heigh)

=> Estes métodos desenharam um retângulo com o canto superior esquerdo nas coordenadas [x, y] e de largura “width” e altura “heigh”.

java.awt.Graphics

3. Desenho de Retângulos Arredondados

public void drawRoundRect(int x, int y, int width, int heigh, int arcW, int arcH)

public void fillRoundRect(int x, int y, int width, int heigh, int arcW, int arcH)

=> Estes métodos desenharam um retângulo com cantos arredondados, situado dentro de um retângulo com o canto superior esquerdo nas coordenadas [x, y] e de largura “width” e altura “heigh”. As ovas que formam os cantos do retângulo têm largura “arcWidth” e altura “arcHeigh”.

4. Desenho de Polígonos

public void drawPolygon(int xPoints[], int yPoints[], int points)

public void fillPolygon(int xPoints[], int yPoints[], int points)

=> Estes métodos desenharam um polígono na cor corrente com o número de pontos “points”, em que a coordenada x de cada ponto está especificada no vetor “xPoints[]” e a coordenada y no correspondente elemento do vetor “yPoints[]”.

java.awt.Graphics

5. Desenho de Ovais: elipses ou círculos

public void drawOval(int x, int y, int width, int height)

public void fillOval(int x, int y, int width, int height)

=> Estes métodos desenharam uma oval (elipse ou círculo) na cor corrente situada dentro de um retângulo com o canto superior esquerdo no ponto [x, y] e de largura “width” e altura “height”.

6. Desenho de Arcos

public void drawArc(int x, int y, int width, int height, int startAngle, int arcA)

public void fillArc(int x, int y, int width, int height, int startAngle, int arcA)

=> Estes métodos desenharam um arco na cor corrente que é parte de uma oval situada dentro de um retângulo com o canto superior esquerdo no ponto [x, y] e de largura “width” e altura “height”. O arco começa no ângulo “startAngle” e estende-se “arcAngle” ângulos.

java.awt.Font

- Pode-se imprimir texto no applet usando a classe **Graphics** em conjunto com a classe **Font**.
- Para desenhar texto o primeiro é necessário criar uma instância da classe **Font**.

```
Font f = new Font ("TimesRoman", Font.BOLD, 14);
```

- Os objetos da classe **Font** representam uma fonte individual, isto é, o nome, estilo (plain, **bold**, *italic*) e tamanho em “points”. Os nomes das fontes são strings representativas da família da fonte, como por exemplo “TimesRoman”, “Courier”, ou “Helvetica”.
- Os estilos das fontes são constantes do tipo inteiro definidas na classe **Font** com as designações **Font.PLAIN**, **Font.BOLD** e *Font.ITALIC*.

java.awt.Color

- Uma cor é representada como uma combinação de vermelho, verde e azul (*Red, Green and Blue* – RGB). Cada componente da cor pode ter um valor entre 0 e 255. Preto é (0,0,0) e branco é (255, 255, 255).
- Na classe `Color` são definidos um conjunto atributos de classe que representam cores padrão, tais como: `Color.white`, `Color.black`, `Color.red`, etc.

```
Color c = new Color(140, 140, 140);
```

- Para se desenhar um objeto ou texto numa determinada cor, é necessário primeiro criar um objeto `Color` e utilizar o método “**setColor(...)**” no objeto `Graphics` desejado.
- Para além de ser possível colocar uma cor corrente para o contexto gráfico, também é possível colocar as cores de “*background*” e “*foreground*” do próprio applet usando os métodos “**setBackground()**” e “**setForeground()**” definidos na classe `java.awt.Component` e que são herdados pela classe.

Exercício

- 1) Escrever no método `paint()` do Applet **AloMundo** as propriedades do computador (versão da JVM e do Sistema operacional) em que o mesmo está sendo executado.
 - Obs: A cor de fundo do applet deverá ser verde; a cor da fonte do texto deverá ser azul; e a fonte Times New Roman 30 Bold.
 - Dica: use a classe **System** e os métodos **setBackground(...)**, **setColor(...)** e **setFont(...)**, respectivamente.



Anotações

Anotações

- É o recurso que possibilita **escrever metadados** (dados sobre outros dados) no código fonte de uma aplicação Java. Essas anotações não afetam o funcionamento do código onde foram declaradas, pois são ignoradas pelo compilador.
- Podem aparecer em qualquer lugar do código e podem ser simples marcadores ou conter elementos que podem receber valores.
- A sua principal utilidade consiste no fato de que as anotações poderão ser utilizadas para criar **diretivas de configuração, controle de versão, validações, testes unitários, tabelas em banco de dados**, etc.
- **Sintaxe**

```
@anotação ("parâmetro")
```

Anotações

- Toda palavra em Java que começa com o símbolo **@** é uma anotação.
- Qualquer tipo de declaração pode possuir uma anotação associada. Por exemplo, classes, métodos, atributos, constantes enum, podem possuir anotações.

- **Exemplos**

```
@Author (  
    name = "Antonio Benedito",  
    date = "04/07/2016"  
)  
class MyClass() { ... }  
  
@SuppressWarnings ("unchecked")  
void myMethod() { ... }
```

Categorias de Anotações

- São três as categorias de anotações:
- **Anotações marcadoras** – são aquelas que não possuem membros.
- **Anotações de valor único** - são aquelas que possuem um único membro.
- **Anotações completas** - são aquelas que possuem múltiplos membros.

```
@Test //marcadora  
@MinhaAnotacao("valor") //valor único  
@Version(major=1,minor=0) //completa
```

Tipos de Anotações

- São dois os tipos de anotações que podem ser utilizados no Java:

(1) Anotações Simples

- Utilizadas para acrescentar significado ao código. Cinco anotações, denominadas '**anotação padrão**', são parte do pacote **java.lang**. São elas: **@Override**, **@SuppressWarnings**, **@Deprecated**, **@SafeVarargs** (novidade Java 7) e **@FunctionalInterface** (novidade Java 8).

(2) Meta-Anotações

- Utilizadas para a criação de anotações. São as anotações das anotações. Pertencem também ao pacote **java.lang.annotation**. São elas: **@Retention**, **@Documented**, **@Target**, **@Inherited** e **@Repeatable** (novidade Java 8).

Anotações Simples

@Override

- É uma anotação marcadora que deve ser usada apenas com métodos. Serve para indicar que o método anotado está sobrescrevendo um método da superclasse.

```
public class Funcionario {  
    protected double salario;  
    public double getSalarioTotal(double bonus)  
    {...}  
    @Override public String toString()  
    {...}  
}
```


Anotações Simples

@Deprecated

- Assim como **@Override**, é também uma anotação marcadora. Esta anotação é utilizada quando é necessário indicar que um método não deveria mais ser usado, ou seja, informa que o método está obsoleto. Diferente de **@Override**, **@Deprecated** deve ser colocada na assinatura do método.

```
public class Funcionario {  
    protected double salario;  
    @Deprecated  
    public double getSalarioTotal(double bonus)  
    {...}  
    @Override public String toString()  
    {...}  
}
```

Anotações Simples

@SuppressWarnings

- Aplicações criadas antes do advento do Java 5 podem ter algum código que gera alertas (*warnings*) durante a compilação com esta versão ou posteriores.
- A resposta está no uso da anotação **@SuppressWarnings**, que permite desligar os alertas de uma parte do código da aplicação – classe, método ou inicialização de variável ou campo – e os *warnings* do restante do código permanecem inalterados.
- Diferente das duas anotações anteriores, **@SuppressWarnings** é uma anotação de valor único, onde o valor é um vetor de String.

Anotações Simples

@SuppressWarnings

```
public class Funcionario {  
    @SuppressWarnings(value={"unchecked","rawtypes"})  
    public static void main(String[] args)  
    {...}  
}
```

Anotações Simples

@SaveVarargs

- A função dessa anotação é informar ao compilador que a operação de conversão forçada de arrays com tipos genéricos é segura, ou seja, que não acontecerá o *heap pollution*.
- Quando usada na declaração de um método com a sintaxe citada, essa anotação desonera as classes que chamam esses métodos da necessidade de utilizarem **@SuppressWarnings**.
- Isso é o que acontece, no JDK 7, com o método **asList()** da classe **Arrays**, como é mostrado a seguir, e também com alguns outros métodos da API.

@SafeVarargs

```
public static <T> List<T> asList(T... a) {  
    return new ArrayList<>(a);  
}
```

Anotações Simples

@FunctionalInterface

- Esta anotação é utilizada para informar que uma interface é explicitamente funcional, isto é, que ela possui **apenas um método abstrato**.
- Interfaces funcionais são o **coração do recurso de Lambda**. O Lambda por si só não existe, e sim expressões lambda, quando atribuídas/inferidas a uma interface funcional.

```
@FunctionalInterface  
  
interface Validador<T> {  
    boolean valida(T t);  
}
```

- As interfaces funcionais definidas no Java 8 fazem parte do pacote **java.util.function**.

Meta-Anotações

@Retention

- As anotações podem estar presentes apenas no código fonte ou no binário de classes ou interfaces. **@Retention** é usada para escolher entre essas possibilidades.
- Ela suporta três valores: **SOURCE**, para indicar que as anotações marcadas não estarão no código binário; **CLASS**, para gravar as anotações no arquivo .class, mas que não estarão disponíveis em tempo de execução; e **RUNTIME**, para indicar que as anotações estarão disponíveis em tempo de execução;

```
@Retention(RetentionPolicy.SOURCE)  
@Retention(RetentionPolicy.CLASS)  
@Retention(RetentionPolicy.RUNTIME)
```

Meta-Anotações

@Target

- Esse tipo de anotação é utilizado para determinar quais são os elementos da classe que poderão ser anotados.
- As possibilidades são:

```
ElementType.ANNOTATION_TYPE (Uma
                               Meta Anotação)
ElementType.CONSTRUCTOR
ElementType.FIELD
ElementType.LOCAL_VARIABLE
ElementType.METHOD
ElementType.PACKAGE
ElementType.PARAMETER
ElementType.TYPE (Enums, Classes
                  ou Interfaces)
```

Meta-Anotações

@Documented

- É uma anotação marcadora utilizada para indicar ao Javadoc a sua inclusão na documentação a ser produzida.

@Inherited

- É uma anotação marcadora utilizada para especificar que as subclasses que estenderem sua classe anotada também serão anotadas.

Criação de Novas Anotações

- É possível a criação das suas próprias anotações, criando uma nova interface com o símbolo @.
- É recurso muito útil ao desenvolvedor que estiver criando uma nova ferramenta ou framework.

```
package anotacoes;  
import java.lang.annotation.*;  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE )  
  
public @interface ITabela {  
    String[]  colunas();  
}
```

- Esta interface é uma Anotação que possui um atributo que define as colunas de uma tabela.

Uso de Novas Anotações

- A nova anotação criada é utilizada pela classe Cliente.

```
package anotacoes;  
@ITabela(colunas = {  
    "nome",  
    "endereco",  
    "cpf"  
})  
public class Cliente  
{  
    private String nome;  
    private String endereco;  
    private String cpf;  
}
```

Tecnologias Java com as suas próprias Anotações

- Hibernate
- EJB
- JPA
- Spring
- JUnit
- CDI
- JSF
- JBoss Seam
- Etc.

Exercícios

- 1) Dada a interface abaixo, quais são as alternativas para eliminar os problemas encontrados?

```
public interface Internet{
    /* conectar @deprecated
    * usar conectarSSL.* /
    @Deprecated
    public void conectar();
    public void conectarSSL();
}

public class InternetApp
    implements Internet
{
    public void conectar() {}
    public void conectarSSL() {}
}
```

Exercícios

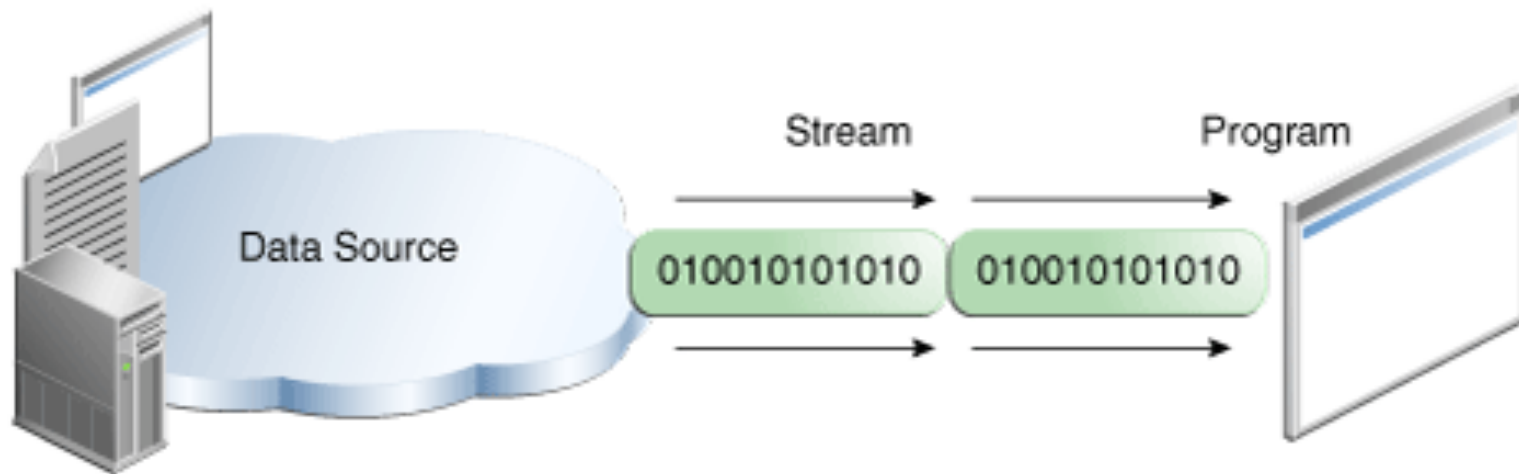
- 2) Criar a Anotação **Copyright** composta por três elementos: autor, data e versao. Aplicar essa anotação à classe **InternetApp**.



Streams de Entrada e Saída

Entrada e Saída

- Toda operação de entrada e saída em uma aplicação Java faz uso de um objeto que identifica um fluxo (*stream*) de informações.
- Um *stream* é uma sequência de dados transmitidos de uma fonte de entrada para um destino de saída e vice-versa.



<http://docs.oracle.com/javase/tutorial/essential/io/streams.html>

Entrada e Saída

- Um **Stream de Entrada** é utilizado para ler os dados a partir de uma fonte, um item de cada vez. Um **Stream de Saída** é utilizado para gravar dados para um destino, um item por vez.
- **InputStream** e **OutputStream** são as classes abstratas definidas no pacote java.io que representam um **Stream de Entrada** e um **Stream de Saída**, respectivamente.
- Os principais *Streams* definidos no Java são:
 - **Byte Stream**
 - **Character Stream**
 - **Buffered Stream**
 - **Data Stream**
 - **Object Stream**

Byte Stream

- É o tipo de fluxo utilizado para realizar a operação de E/S de bytes.
- As classes **FileInputStream** e **FileOutputStream** representam este tipo de fluxo.

```
FileInputStream in = null;
FileOutputStream out = null;
try {
    in = new FileInputStream("fe.txt");
    out = new FileOutputStream("fs.txt");
    int c;
    while ((c = in.read()) != -1)
        out.write(c);
}
```

Character Stream

- A plataforma Java armazena os caracteres utilizando a notação Unicode. Um *Character Stream* automaticamente converte o formato utilizado para o formato local, facilitando a internacionalização da aplicação.
- Todas as classes de *Character Stream* são descendentes de **Reader** e **Writer**.

```
FileReader in = null;
FileWriter out = null;
try {
    in = new FileReader("fe.txt");
    out = new FileWriter("fs.txt");
    int c;
    while ((c = in.read()) != -1)
        out.write(c);
}
```

Buffered Stream

- Os fluxos vistos até então não fazem uso de uma memória auxiliar (buffer) para armazenar o que está sendo enviado/recebido.
- O Java implementa este tipo de fluxo para oferecer este tipo de operação.

```
BufferedReader in = new BufferedReader  
    (new InputStreamReader(System.in));  
String monitor="";  
System.out.println("Digite algo");  
try {  
    monitor = teclado.readLine();  
}  
System.out.println("foi digitado:"+monitor);
```

Data Stream

- É o tipo de fluxo que oferece suporte ao envio/recebimento de tipos de dados (boolean, char, byte, short, int, long, float, e double), bem como de Strings também.
- Todos os *Data Streams* implementam a interface **DataInput** ou **DataOutput**.

```
static final double[] precos = {19, 9, 15};
static final int[] unidades = {12, 8, 13};
static final String[] descs = {"TS", "pin", "boné"};
out = new DataOutputStream
    (new BufferedOutputStream(
        new FileOutputStream("fs.txt")));
for (int i = 0; i < precos.length; i++) {
    out.writeDouble(precos[i]);
    out.writeInt(unidades[i]);
    out.writeUTF(descs[i]);
}
```

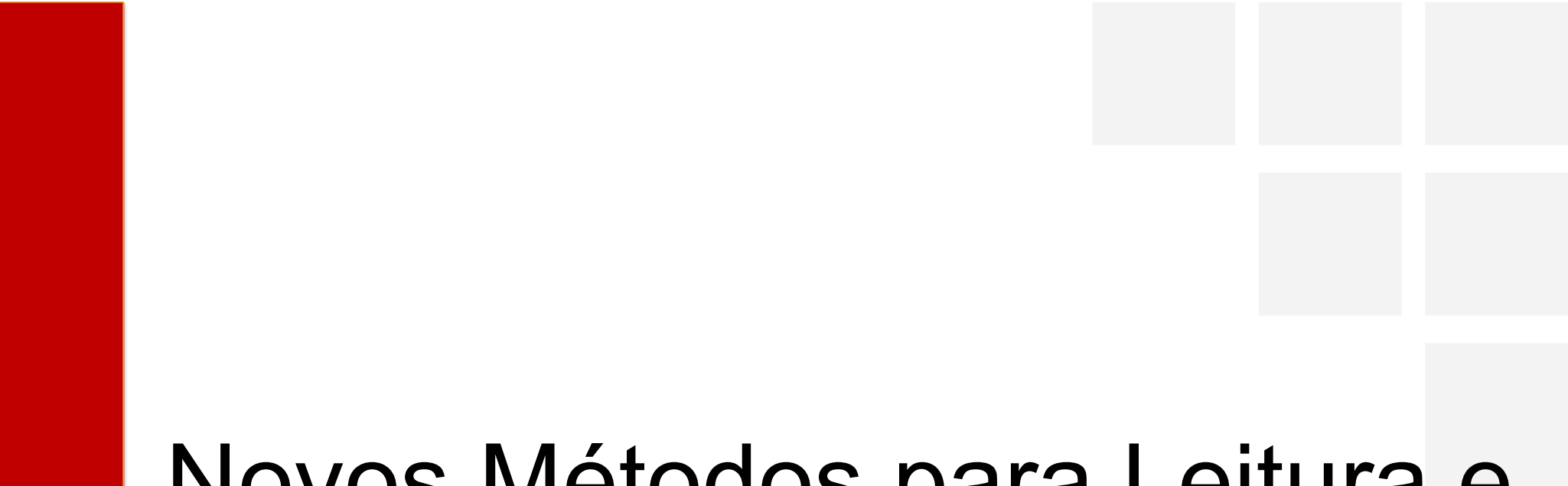
Object Stream

- É o tipo de fluxo que oferece suporte ao envio/recebimento de Objetos, cuja classe implemente a interface *Serializable*.
- As classes *Object Stream* são **ObjectInputStream** e **ObjectOutputStream**.

```
Object ob = new Object();  
//escrita  
out.writeObject(ob);  
  
//leitura  
Object ob2 = in.readObject();
```

Exercícios

- 1) Escrever a classe **StreamApp** que leia um arquivo texto (**dados.txt**) e informe na tela o número de caracteres totais existentes nesse arquivo.
- 2) Na classe **StreamApp**, criar um método para contar o número de vezes que um determinado caracter (por exemplo 'a'), informado pelo usuário em tempo de execução, aparece no arquivo texto (**dados.txt**).



Novos Métodos para Leitura e Escrita em Arquivos

LEITURA E GRAVAÇÃO EM ARQUIVOS

- **Definição**



- O Java 11 facilitou enormemente a tarefa de ler/escrever String em arquivos com a criação dos métodos `readString()` e `writeString()`.

- **Leitura**

```
String texto = Files.readString(Path.of("arquivo"));
```

- **Escrita**

```
Files.writeString(Path.of("arquivo"), novoConteudo);
```

Entrada e Saída de Dados com as classes Scanner e Formatter

Classes Scanner e Formatter

- A classe **Scanner** implementa as operações de entrada de dados pelo teclado.
- A classe **Formatter** implementa as operações de saída dos dados gerados pela aplicação.

```
Formatter fmt = new Formatter(System.out);  
Scanner scn = new Scanner(System.in);  
int n1 = scn.nextInt();  
fmt.format("n1: %d", n1);
```

- O método principal (**format**) é muito similar à função **printf(...)** da linguagem 'C'.

```
fmt.format(String fmt, Object ... args);
```

Classes Scanner e Formatter

- Os principais caracteres de formatação estão apresentados abaixo:

<code>%c</code>	caractere simples
<code>%d</code>	decimal
<code>%e</code>	notação científica
<code>%f</code>	ponto flutuante
<code>%o</code>	octal
<code>%s</code>	cadeia de caracteres
<code>%u</code>	decimal sem sinal
<code>%X</code>	hexadecimal

- Para cada um dos tipos primitivos, há um método **nextXxx()** correspondente.

```
int n = ler.nextInt();  
float preco = ler.nextFloat();  
double salario = ler.nextDouble();  
String palavra = ler.next();
```

Classes Scanner e Formatter

- Tanto a classe **Scanner** como a **Formatter** pertencem ao pacote **java.util**.
- A classe Scanner divide em substrings (*tokens*) qualquer dado entrado. Estes são separados por delimitadores.
- Se a string não puder ser interpretada com o tipo especificado, a exceção **InputMismatchException** é anunciada.
- Há também um conjunto de métodos **hasNextXxx()**, como por exemplo **hasNextInt()**, que retornam verdadeiro ou falso, de acordo com o tipo de *token* passado.
- A classe **Formatter** fornece vários construtores, cada qual com um tipo de parâmetro específico.

```
Formatter(File file)
Formatter(OutputStream os)
Formatter(String fileName)
Formatter(PrintStream ps)
```

Classes Path e Files

- O pacote **java.nio.file** fornece a classe Path para manipular o caminho em um determinado sistema de arquivos.

```
// Sintaxe Microsoft Windows  
Path path = Paths.get("C:\\home\\joe\\foo");
```

- Também fornece a classe Files composta por vários métodos para manipulação de arquivos, tais como: **criação**, **cópia** e **exclusão** de arquivos.

```
Files.createFile(file);  
Files.delete(path);  
Files.copy(source, target, REPLACE_EXISTING);  
Files.move(source, target, REPLACE_EXISTING);  
Files.createDirectory(path);
```

Exercícios

- 1) Escrever a classe **ScannerApp** que leia quatro notas, calcule a sua média aritmética e imprima na tela o seu resultado com duas casas decimais.
- 2) Escrever a classe **ScannerAppv2** que leia o arquivo **numeros.txt** (8.5 32767 3.14159 1000000.1) e calcule a soma dos seus valores, imprimindo o resultado na tela.
- 3) Escrever a classe **BackupApp** que realiza o backup de todos os arquivos **.java** existentes no pacote **unidade2**.