📌 12.1 Entrada e Saída em Java usando Arquivos

✅ 12.1.1 Streams

- **Streams** são fluxos de dados unidirecionais:
    - **Entrada (InputStream):** dados fluem **do dispositivo para o programa**.
    - **Saída (OutputStream):** dados fluem **do programa para o dispositivo**.
- **Exemplos de dispositivos**: teclado, monitor, arquivos, rede, etc.

🔄 Tipos de Stream

- **De bytes:** manipula dados brutos (modo binário). Ex: imagens, vídeos, etc.
- **De caracteres:** manipula texto (modo texto). Ex: arquivos `.txt`.

📦 Bufferizados x Não Bufferizados

- **Bufferizados:** usam uma área de memória intermediária (mais rápidos).
- **Não bufferizados:** acessam diretamente os dispositivos (mais simples, porém mais lentos).

📁 Pacote `java.io`

- Fornece as classes essenciais para manipulação de arquivos e dispositivos com streams.

✅ 12.1.2 `InputStream` e `OutputStream`

Essas são classes **abstratas base** para leitura e escrita de dados **em bytes**.

🟦 Classe `InputStream` (leitura)

Método	Descrição

`int read()`	Lê um único byte, retorna como inteiro. Retorna `-1` no fim do stream.
`int read(byte[] b)`	Lê múltiplos bytes e armazena no array `b`.
`int read(byte[] b, int off, int len)`	Lê até `len` bytes e coloca no array a partir da posição `off`.
`void close()`	Fecha a stream (libera recursos).
`int available()`	Retorna número de bytes disponíveis sem bloqueio.
`long skip(long n)`	Pula os próximos `n` bytes da stream.
`boolean markSupported()`	Verifica se a stream suporta marcar/resetar posição.
`void mark(int readlimit)`	Marca a posição atual para voltar depois.
`void reset()`	Volta para a posição marcada anteriormente.

🟩 Classe `OutputStream` (escrita)

Método	Descrição
`void write(int b)`	Escreve um único byte na stream.
`void write(byte[] b)`	Escreve todos os bytes do array.
`void write(byte[] b, int off, int len)`	Escreve parte dos bytes do array (de `off` até `off+len`).
`void flush()`	Força a gravação dos dados no destino (útil com buffers).
`void close()`	Fecha a stream e libera recursos.

💡 Dica prática

Para ler e escrever arquivos em Java moderno, muitas vezes usamos classes como:

- `FileInputStream`, `BufferedInputStream`
- `FileOutputStream`, `BufferedOutputStream`

- Ou até as classes de **modo texto**: `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`.

Exemplo simples de uso:

```
try (FileOutputStream out = new FileOutputStream("saida.txt")) {
    String texto = "Olá, mundo!";
    out.write(texto.getBytes()); // escreve os bytes da string
}
```

Ótimo! Você está estudando um ponto fundamental da programação Java: **entrada e saída com arquivos usando `FileInputStream` e `FileOutputStream`**, incluindo a **gravação e leitura de objetos serializados**.

Abaixo, explico os conceitos e forneço os exemplos de código mencionados (64, 65 e 66), com comentários para facilitar o entendimento.

✅ 12.1.3 - Manipulação de arquivos com FileInputStream e FileOutputStream

🔹 Resumo das Classes

Classe	Função
`FileInputStream`	Leitura de arquivos (modo byte)
`FileOutputStream`	Escrita de arquivos (modo byte)

📦 Construtores e métodos principais

Classe	Construtor / Método	Descrição
`FileInputStream`	`FileInputStream(String n)`	Abre arquivo para leitura
	`int read()`	Lê um byte

	`int read(byte[] b)`	Lê bytes no array
	`long skip(long n)`	Pula `n` bytes
	`int available()`	Retorna bytes disponíveis
	`void close()`	Fecha a stream
`FileOutputStream`	`FileOutputStream(String n)`	Abre arquivo para escrita (sobrescreve)
	`FileOutputStream(String n, boolean append)`	Se `true`, adiciona no fim
	`void write(int b)`	Grava um byte
	`void write(byte[] b)`	Grava um array de bytes
	`void close()`	Fecha a stream

💾 Exemplo de gravação de objeto em arquivo

📄 Código 64: Classe Imovel

```
import java.io.Serializable;

public class Imovel implements Serializable {
    private String endereco;
    private double preco;

    public Imovel(String endereco, double preco) {
        this.endereco = endereco;
        this.preco = preco;
    }

    public String getEndereco() {
        return endereco;
    }

    public double getPreco() {
        return preco;
    }

    @Override
    public String toString() {
        return "Endereço: " + endereco + ", Preço: R$ " + preco;
```

```
    }
}
```

> **Observação:** A classe implementa `Serializable`, necessário para gravar objetos em arquivos.

💾 Código 65: Classe GravarImovel

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class GravarImovel {
    public static void main(String[] args) {
        Imovel imovel = new Imovel("Rua das Flores, 123", 350000.00);

        try (FileOutputStream fos = new FileOutputStream("imovel.dat");
             ObjectOutputStream oos = new ObjectOutputStream(fos)) {

            oos.writeObject(imovel);
            System.out.println("Imóvel gravado com sucesso!");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

> Aqui usamos `ObjectOutputStream` para serializar e gravar o objeto `Imovel`.

📂 Exemplo de leitura do objeto gravado

📄 Código 66: Classe LerImovel

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class LerImovel {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("imovel.dat");
             ObjectInputStream ois = new ObjectInputStream(fis)) {
```

```
            Imovel imovel = (Imovel) ois.readObject();
            System.out.println("Imóvel lido: " + imovel);

        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Este exemplo lê o objeto serializado do arquivo e o reconstrói em memória.

🧠 Conclusão

- `FileInputStream` e `FileOutputStream` trabalham com arquivos em **nível de byte**.
- Para objetos, é necessário usar `ObjectOutputStream` / `ObjectInputStream` junto com as streams de arquivo.
- **Serializable** é obrigatório para gravar objetos em arquivos.