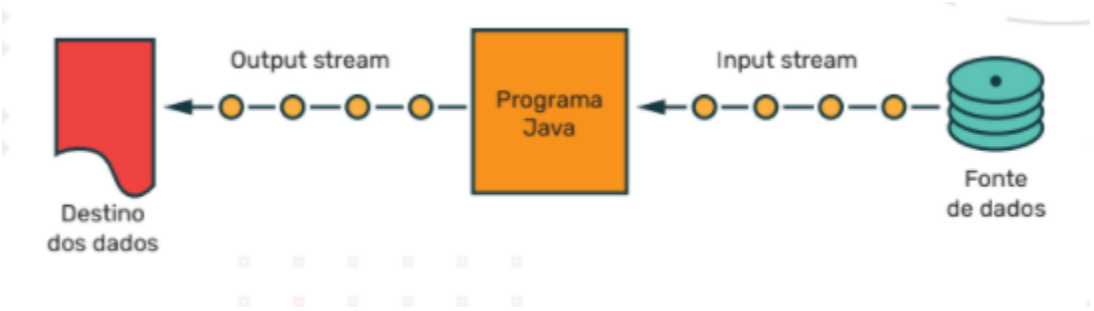
Certified Tech Developer
The Ultimate Degree

Fluxos de dados

Em Java, assim como em outras linguagens, a entrada e a saída de dados em nosso programa são conhecidos como fluxo de dados ou stream. Eles podem ser de entrada ou de saída. Para entender a "direção" do fluxo, tomamos como ponto de vista o código que está sendo executado: o fluxo de saída ocorre quando obtemos os dados e estes "saem" da plataforma, quando por exemplo, são gravados em disco, enviados pela rede, etc. No fluxo de entrada se recebe dados que "chegam" da plataforma, quando por exemplo, realizamos a leitura de um arquivo em disco ou dispositivo, e assim por diante

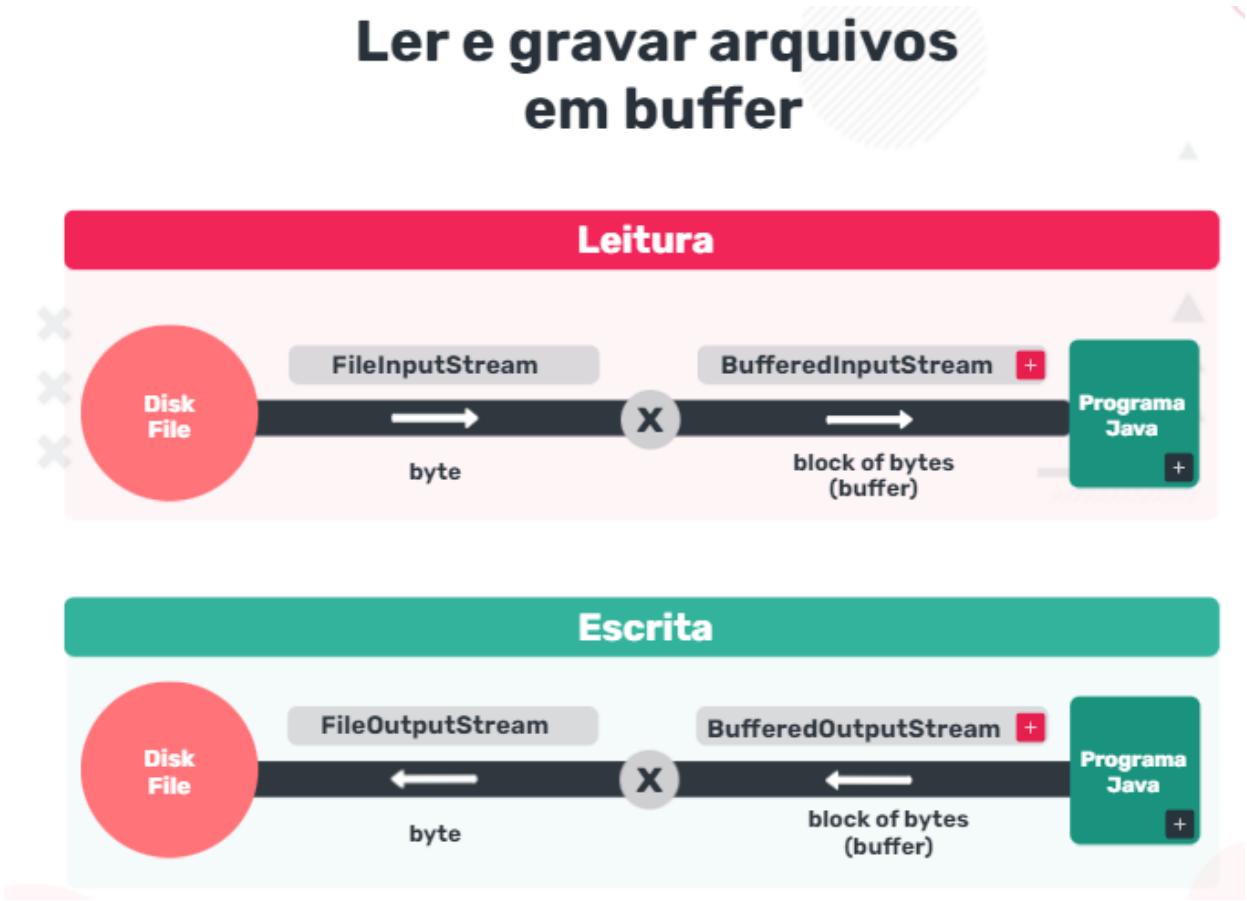


Buffers

Todos os fluxos de entrada e saída podem funcionar de duas maneiras: byte-a-byte ou por "lote". Em outras palavras, se um arquivo de 1024 bytes tiver que ser lido, podemos acessá-lo 1024 vezes para ler cada byte ou acessá-lo 8 vezes para ler em lotes de 128 bytes. Para isso, fluxos binários ou de caracteres estabelecem subclasses "Buffered", podendo ser visto nos diagramas apresentados anteriormente. Essas classes disponibilizam métodos que permitem escolher o tamanho dos lotes para leitura e gravação. Se o tamanho não for definido, essas classes assumem um tamanho padrão.

A escolha de usar um fluxo com "buffer" ou não, depende de cada aplicação. Enquanto os buffers aceleram as operações de leitura, eles consomem mais memória. Cada lote lido de uma vez é armazenado na memória até que algo seja feito com ele, como gravá-lo em outro lugar. Por outro lado, a não utilização de buffers consome menos memória, porque um lote temporário não é armazenado na memória, mas implica que, a cada byte lido, ocorra uma chamada para o sistema operacional ou plataforma, intensificando o uso da CPU.

Dito isso, o uso de streams com buffer tem algumas vantagens para o programador: Por exemplo, o BufferedInputStream permite ler blocos de bytes, que possibilita ler um arquivo com muita facilidade. Caso contrário, você teria que ler caractere por caractere e detectar o "retorno de carro" para saber quando cada linha termina. Vamos prosseguir para aprender mais!



Métodos de BufferedInputStream

Método	Descrição
int available ()	Retorna o número estimado de bytes disponíveis para leitura.
void close ()	Fecha o BufferedInputStream.
void mark (int readLimit)	Marca a posição atual para ler no fluxo de entrada.
boolean markSupported ()	Verifica se o fluxo é compatível com os métodos mark() e reset().
int read ()	Lê um byte de dados do fluxo de entrada.
int read (byte [] b)	Lê o byte especificado da matriz de entrada.
int read (byte [] b, int off, int len)	Lê os bytes de dados da matriz, começando na posição especificada.
byte [] readAllBytes ()	Lê todos os bytes restantes do fluxo de entrada.
byte [] readNBytes (int len)	Lê até o número especificado de bytes.
int readNBytes (bytes [] b, int off, int len)	Lê até o comprimento especificado de bytes da matriz de bytes, começando na posição de deslocamento.
long skip (long n)	Desconsidera ou descarta o número especificado de bytes durante a operação de leitura.
void skipNBytes (long n)	Salta ou descarta até o número especificado de bytes durante a operação de leitura.
long transferTo (OutputStream out)	Lê todos os bytes do fluxo de entrada e os grava no fluxo de saída especificado na mesma ordem.

Para ler o conteúdo de um arquivo, devemos instanciar um objeto FileInputStream e informá-lo como parâmetro para o construtor da instância do objeto do tipo BufferedInputStream. Então, vamos realizar a leitura (.read()) até que não retorne mais conteúdo (-1).

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class TestReadFile {

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

   FileInputStream fi = new FileInputStream("InputFile.txt");
   BufferedInputStream bi = new BufferedInputStream(fi);

   try {
     int i;
     while((i=bi.read()) != -1) {
       System.out.print((char)i);
     }

     bi.close();
     fi.close();
   } catch (IOException e) {
     e.printStackTrace();
   }

 }
}
```

Métodos de BufferedOutputStream

Método	Descrição
void close ()	Fecha o fluxo de saída e libera todos os recursos.
void flush ()	Libera o BufferedOutputStream e grava todos os dados restantes no fluxo de saída.
void write (byte [] b)	Escreve os bytes no fluxo de saída.
void write (int byte)	Grava o byte especificado no fluxo de saída.
void write (byte [] b, int off, int len)	Escreve o comprimento especificado de bytes da matriz no fluxo de saída, começando na posição de deslocamento.

Para escrever conteúdo em um arquivo, devemos instanciar um objeto do tipo FileOutputStream e, em seguida, informá-lo como parâmetro para o construtor da instância do objeto do tipo BufferedOutputStream. Posteriormente, escrevemos (.write()) informando o conteúdo desejado, neste caso um número.

```
import java.io.BufferedOutputStream
import java.io.FileNotFoundException
import java.io.FileOutputStream
import java.io.IOException
public class WriteBufferedFile {
 public static void main(String[]
args) throws FileNotFoundException {
   FileOutputStream fo
= new FileOutputStream("OutputFile.txt");
   BufferedOutputStream bo
= new BufferedOutputStream(fo);
   byte b = 65;
   try {
     bo.write(b);
     System.out.println("El byte fue escrito
correctamente");
     bo.close();
    fo.close();
   } catch (IOException e) {
     e.printStackTrace();
   }

 }
}
```