devmedia.com.br

Classe InputStream e Outputstream em Java

22-30 minutos

fechar
Guia Linguagem Java

favorito (12) imprimir anotar marcar como lido dúvidas?
Você precisa estar logado para dar um feedback.

Clique aqui para efetuar o login
Confirmar voto

Veja neste artigo como funcionam as classes InputStream e OutputStream em Java.

Neste artigo veremos algo muito usado na maioria dos softwares mas que ainda é alvo de muitas dúvidas devido à grande quantidade de funcionalidades apresentadas por estes conceitos: O InputStream e OutputStream em Java.

Entenda que existe o conceito de "**Input**" e "**Output**" para qualquer dos casos tratados, ou seja, tudo tem um destino e uma fonte. Um programa que precise ler algum dado de algum local (uma fonte) precisa de um **InputStream** ou um Reader, por outro lado um programa que precise escrever um dado em algum local (destino) precisa de um **OutputStream** ou um Writer.

Você verá muito essa palavra "Stream" que é um fluxo de dados, seja para leitura ou para escrita. Imagine um Stream como uma conexão com uma fonte ou destino de dados, onde esses dados podem ser passados via byte ou character. Por exemplo, um arquivo de texto pode representar um Stream, onde o seu programa irá ler esse Stream via byte ou character usando InputStream ou algum Reader.

InputStream

Vimos então que o **InputStream** faz parte da leitura de dados, ou seja, está conectado a alguma fonte de dados: arquivo, conexão de internet, vídeo e etc. O InputStream nos possibilita ler esse Stream em byte, um byte por vez. Acontece que se olharmos na classe **InputStream** veremos que ela é abstrata e nós somos obrigados a usar alguma outra classe que a implemente para fazer uso dos seus recursos. Algumas delas: *FileInputStream*, *BufferedInputStream*, *DataInputStream* e etc. Vejamos um exemplo na **Listagem 1.**

Listagem 1. Exemplo de InputStream com FileInputStream

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

  public class MyApp {

          /**
           * @param args
           * @throws
           */
          public static void main(String[] args) {
                InputStream inputstream;
                try {
                      inputstream = new
FileInputStream("/tmp/input_text");

                      int data = inputstream.read();
                      while (data != -1) {
                            System.out.println(data);

                                  data =
inputstream.read();

                      }
                      inputstream.close();
                } catch (FileNotFoundException e1) {
                      // TODO Auto-generated catch
block
                      e1.printStackTrace();
                } catch (IOException e) {
                      // TODO Auto-generated catch
```

```
block
                        e.printStackTrace();
                }

        }

  }
```

Bom, o que fizemos acima foi criar uma instância de *FileInputStream* passando como argumento a localização do Stream quegostaríamosde ler. O método read() do **InputStream** retorna um valor inteiro que contém obyte correspondente que foi lido. Enquanto este valor lido for diferente de -1 significa que a leitura do Stream ainda não terminou. Então nós lemos repetidas vezes que o valor -1 seja atingido assim saberemos que a leitura terminou.

De posse do valor em byte você pode converte-lo para char afim de mostrar uma informação mais relevante, caso seja necessário.

Listagem 2. Convertendo a leitura do InputStream para char

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

  public class MyApp {

          /**
           * @param args
           * @throws
           */
          public static void main(String[] args) {
                InputStream inputstream;
                try {
                        inputstream = new
FileInputStream("/tmp/input_text");

                        int data = inputstream.read();
                        while (data != -1) {
                               System.out.println((char)
data);

                                        data =
```

```
inputstream.read();

                        }
                        inputstream.close();
                } catch (FileNotFoundException e1) {
                        // TODO Auto-generated catch
block
                        e1.printStackTrace();
                } catch (IOException e) {
                        // TODO Auto-generated catch
block
                        e.printStackTrace();
                }

        }

  }
```

Perceba na **Listagem 2** que nós adicionamos o cast “char” ao System.out.println() assim nós estamos convertendo explicitamente o valor que antes era byte para um char.

Vimos que o *FileInputStream* nos permite ler um arquivo qualquer e retornar os dados em byte. Temos ainda o *BufferedInputStream* que diferente do *FileInputStream*, que lê byte a byte, este lê um bloco inteiro de uma só vez, agilizando o processamento de leitura no disco.

Um exemplo prático: Se o seu arquivo possui 32768 bytes, para que um *FileInputStream* possa ler ele por completo, ele precisará fazer 32768 chamadas ao Sistema Operacional. Com um BufferedInputStream você precisará de apenas quadro chamadas, isso porque o *BufferedInputStream* armazena 8192 bytes em um buffer e os utiliza quando precisa. Resumindo, você deve usar o BufferedInputStream como um wrapper para o FileInputStream quando desejar ganhar mais velocidade.

Listagem 3. Usando BufferedInputStream como wrapper para FileInputStream

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

  public class MyApp {
```

```
	/**
	 * @param args
	 * @throws
	 */
	public static void main(String[] args) {
		InputStream inputstream;
		try {
			inputstream = new
BufferedInputStream(
			new FileInputStream("/tmp
/input_text"));

			int data = inputstream.read();
			while (data != -1) {
				System.out.println((char)
data);

					data =
inputstream.read();

			}
			inputstream.close();
		} catch (FileNotFoundException e1) {
			// TODO Auto-generated catch
block
			e1.printStackTrace();
		} catch (IOException e) {
			// TODO Auto-generated catch
block
			e.printStackTrace();
		}

	}

  }
```

Perceba no exemplo da **Listagem 4** que encapsulamos o FileInputStream dentro do BufferedInputStream para tornar nosso processo de leitura mais rápido, visto que o BufferedInputStream armazena um buffer de informações, como o próprio nome já sugere.

Além do método read() padrão presente no InputStream, ainda temos duas

alternativas, como veremos nas **Listagens 4 e 5.**

Listagem 4. Alternativas para o método read()

```
int read(byte[])
int read(byte[], int offset, int length)
```

O primeiro método read(byte[]) possibilita a leitura de uma quantidade certa de bytes, em vez de ler byte a byte. Isso torna o processo mais rápido e eficaz quando há uma grande quantidade de dados a serem lidos. O segundo método também ler uma certa quantidade de bytes mas diferente do primeiro ele possui o campo offset e length que dizem respeito ao início e fim da leitura respectivamente.

Listagem 5. Usando read(byte[])

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

public class MyApp {

        /**
         * @param args
         * @throws
         */
        public static void main(String[] args) {
              InputStream inputstream;
              try {
                     inputstream = new
BufferedInputStream(
                     new FileInputStream("/tmp
/input_text"));
                     byte[] dataAsByte = new byte[2];
                     inputstream.read(dataAsByte);
                     for(int i =0; i < 2 ; i++){
                            System.out.println((char)
dataAsByte[i]);
                     }

                     inputstream.close();
              } catch (FileNotFoundException e1) {
                     // TODO Auto-generated catch
```

```
block
                        e1.printStackTrace();
                } catch (IOException e) {
                        // TODO Auto-generated catch
block
                        e.printStackTrace();
                }

        }

  }
```

O que fizemos na **Listagem 5** foi criar um array de byte chamado dataAsByte com tamanho igual a 2, depois passamos ele como parâmetro para o método read() e o mesmo saberá que deverá retornar dois bytes e armazenar dentro de dataAsByte. Por fim, nós fazemos um laço for para mostrar o valor destes dois bytes retornados.

Mark e Reset

Dois métodos muito importantes ainda no InputStream, são os métodos mark() e reset(). São métodos simples de entender e muito úteis. O método mark() serve para criar uma marcação em determinada parte da leitura do stream, assim a qualquer hora podemos voltar a leitura daquele ponto que foi marcado, usando o método reset().

Listagem 6. Entendendo o mark() e reset()

```
  import java.io.BufferedReader;
  import java.io.IOException;
  import java.io.StringReader;

  public class MyApp {

        /**
         * @param args
         * @throws
         */
        public static void main(String[] args) {
                String s ="ABCDE";
              StringReader sr = null;
              BufferedReader br = null;

              try{
```

```
                    sr = new StringReader(s);

                    br = new BufferedReader(sr);

                    System.out.println((char)br.read());
                    System.out.println((char)br.read());

                    //marco este ponto
                    br.mark(0);
                    System.out.println("mark() invoked");

                    System.out.println((char)br.read());
                    System.out.println((char)br.read());

                    // volto para o ponto marcado
                    br.reset();
                    System.out.println("reset()
invoked");

                    System.out.println((char)br.read());
                    System.out.println((char)br.read());

                } catch (Exception e) {

                    // exception occurred.
                    e.printStackTrace();
                }finally{

                    // releases any system resources
associated with the stream
                    if(sr!=null)
                       sr.close();
                    if(br!=null)
                              try {
                                     br.close();
                              } catch (IOException e) {
                                     // TODO Auto-
generated catch block

e.printStackTrace();
                              }
                }
```

```
            }
    }
```

Na **Listagem 6** nós trabalhamos com três variáveis: s, sr e br. A variável 's' armazenará a String ABCDE que nós usaremos como exemplo de leitura, a variável 'sr' servirá de entrada para o a nossa variável 'br' que é o BufferedReader().

Depois de criarmos o BufferedReader passando a string “ABCDE” começamos a leitura, chamando o método read() duas vezes, o que irá nos retornar “A” e logo em seguida “B”, e o ponteiro de leitura agora está posicionado na letra “C”. Chamamos o método mark() para marcar a letra “C”, assim voltamos a ela quando desejarmos.

Continuamos a leitura chamando o read() mais duas vezes, o que nos retorna “C” e logo em seguida “D”. Depois chamamos o método reset() que retorna o ponteiro para a letra “C” mesmo estando atualmente na letra “D”, depois de executar o reset() nós realizamos mais duas chamadas ao método read() que nos retorna “C” e “D”, perceba que nem chegamos a letra “E” por estamos trabalhando com o mark() e reset(), voltando assim sempre ao ponto que desejamos.

O uso de ambos os métodos é muito visto quando precisamos ficar retornando para determinados pontos específicos da leitura, algo que não seria possível sem um marcador. Imagine, por exemplo, que você deseje certificar-se que após determinado caractere nada foi mudado, ou seja, sempre continuará a sequência que você deseja, para isso toda vez que for escrito algo você pode voltar a determinado ponto e certificar-se que nada foi alterado e ser for alterado você pode lançar alguma exceção dizendo que aquela alteração não é permitida naquele ponto. Óbvio que existem muitas aplicações para o uso destes métodos e quando você se deparar com uma dessas você já saberá como recorrer ao local correto.

OutputStream

Vimos tudo que é necessário para fazer a leitura de uma Stream usando InputStream juntamente com suas subclasses (FileInputStream, BufferedInputStream e etc). Agora veremos o processo de escrita neste Stream, ou seja, o OutputStream é capaz de enviar dados a um determinado Stream, ao contrário do InputStream que faz a leitura do mesmo. Lembre-se que ao falarmos de “Stream” não estamos tratando especificamente de texto mas qualquer tipo de dado, usamos a leitura de texto apenas para facilitar o entendimento, mas outros métodos poderiam ser utilizados.

Assim como o InputStream, o OutputStream é uma classe Abstrata que precisa de uma implementação concreta, alguma de suas implementações são: BufferedOutputStream, FileOutputStream e etc. Se no InputStreamtínhamoso método read(), agora no OutputStream temos o método write().

Listagem 7. Escrevendo em arquivo com OutputStream

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

  public class MyApp {

          /**
           * @param args
           * @throws
           */
          public static void main(String[] args) {
                try {
                        OutputStream output = new
FileOutputStream("/tmp/input_text");
                        String s = "ABCDE";
                        int count = s.length()-1;

                        while (count >= 0) {


output.write(s.charAt(count));

                                count--;
                        }
                        output.close();
                } catch (IOException e) {
                        // TODO Auto-generated catch
block
                        e.printStackTrace();
                }

          }
  }
```

Veja na **Listagem 7** que usamos o FileOutputStream como classe concreta

para implementar o OutputStream, assim passamos como argumento o arquivo “/tmp/input_text” que é local onde iremos escrever a String ABCDE. Fazemos um laço while que irá percorrer toda a string e escrever no arquivo input_text através do método write().

Assim como no InputStream, onde tínhamos read(), read(byte[] bytes) e read(byte[] bytes, int offset, int length), nós temos o write(byte), write(byte[] bytes) e write(byte[] bytes, int offset, int length). Em vez de escrevermos byte a byte no arquivo, nós podemos enviar um array de bytes com o método acima apresentado, igualmente como fizemos na leitura usando o read(byte[] bytes).

Listagem 8. Usando write(byte[] bytes)

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

  public class MyApp {

          /**
           * @param args
           * @throws
           */
          public static void main(String[] args) {
                try {
                        OutputStream output = new
FileOutputStream("/tmp/input_text");
                        byte[] bytes = new byte[]
{'A','B', 'D', 'E'};
                        int count = bytes.length;

                        while (count >= 0) {

                                output.write(bytes);

                                count--;
                        }
                        output.close();
                } catch (IOException e) {
                        // TODO Auto-generated catch
block
                        e.printStackTrace();
```

```
                }

        }
  }
```

Na **Listagem 8** vamos passar quatro vezes dentro do laço while e em cada uma dessas iterações iremos escrever “ABCDE” no arquivo. No final teremos várias repetições de “ABCDE”.

flush()

Quando alguma escrita é feita em um arquivo em disco, pode ser que este dado ainda não tenha sido de fato escrito no disco, esteja em algum local na memória esperando o momento exato para ser gravado. O método flush() força que este dado seja escrito imediamente no disco ou em qualquer outro local que você esteja tentando gravar.

close()

O método close() deve ser chamado para fechar a escrita do arquivo e evitar que este fique aberto. Como várias exeções podem ocorrer durante a escrita do mesmo, e o fechamento do arquivo deve ser sempre garantido, o ideal é que o close() fique no bloco finally de uma try-catch.

Listagem 9. Usando close() no finally

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

  public class MyApp {

        /**
         * @throws IOException
         * @param args
         * @throws
         */
        public static void main(String[] args) throws
IOException {
              OutputStream output = null;
              try {
                    output = new
FileOutputStream("/tmp/input_text");
                    byte[] bytes = new byte[]
```

```
{'A','B', 'D', 'E'};
                        int count = bytes.length;

                        while (count >= 0) {

                                output.write(bytes);

                                count--;
                        }

                }finally{
                        if (output != null){
                         output.close();
                        }
                }

        }
  }
```

Na **Listagem 9** Verificamos se a variável output é diferente de nulo então fechamos o Stream.

Ambos os conceitos de leitura e escrita em Stream são muito úteis quando tratamos de comunicação remota, por exemplo Socket. O OutputStream é usado para enviar dados do cliente ao servidor, enquanto o InputStream é usado para ler os dados que chegaram no servidor. Vejamos um exemplo prático disso na **Listagem 10.**

Listagem 10. Enviando dados com OutputStream usando Socket

```
  import java.io.IOException;
  import java.io.OutputStream;
  import java.net.Socket;

  public class MyApp {

        private static Socket socket;

        public static void main(String[] args) throws
IOException {
                String s = "enviando dados com Socket
usando OutputStream";
                OutputStream outputStream =
socket.getOutputStream();
```

```
            outputStream.write(s.getBytes());
            outputStream.flush();
        }
  }
```

É óbvio que no exemplo acima nós abstraímos a conexão com o servidor para não tornar a listagem mais complexa e perder o foco do artigo, mas o que queremos chamar atenção é que através do objeto Socket nós usamos o método getOutputStream() que nos retorna uma instância de OutputStream onde podemos escrever o que quisermos e o servidor irá receber através do InputStream.

Vejamos nosso servidor usando InputStream na **Listagem 11.**

Listagem 11. Usando InputStream com Socket

```
  import java.io.IOException;
  import java.net.ServerSocket;
  import java.net.Socket;
  import java.util.Scanner;

  public class MyApp {

        public static void main(String[] args) throws
IOException {
            ServerSocket servidor = new
ServerSocket(12345);

            Socket cliente = servidor.accept();

            Scanner entrada = new
Scanner(cliente.getInputStream());
            while (entrada.hasNextLine()) {

System.out.println(entrada.nextLine());
            }

            entrada.close();
            servidor.close();
        }
  }
```

Perceba agora que em vez de usar o getOutputStream, usamos o getInputStream() que nos retorna exatamente o que o cliente está enviando “do outro lado”. Lembre-se que o conceito de Stream é um bloco

genérico para algum tipo de dados, podendo ele ser texto, vídeo, imagem e etc, realmente não importa. O que nos importa é entender as duas pontas da conexão o InputStream e o OutputStream.

Tanto o InputStream como o OutputStream são usados para leitra e escrita de dados usando bytes, diferente de outros tipos como o InputStreamReader que usam caracteres. Ambos são muito importantes e utilizados em diversos casos, como o caso do Socket que mostramos na seção final.

Mostramos neste artigo o uso em detalhes de ambas as classes e seus métodos, como aplicá-los e qual o objetivo de cada um. Você, caro leitor, verá que existem ainda muitas outras aplicações onde estas são exigidas e podem fazer grande diferença em questão de desempenho e eficácia. Como demonstramos logo no início que o BufferedInputStream é muito mais rápido que o FileInputStream. O fato de o Java tratar o dado como um Stream e não como um texto, vídeo ou qualquer outro tipo de arquivo em específico, serve para deixar este o mais genérico possível, não importante o tipo de fonte ou destino ao qual pretendemos trabalhar.

O problema começa a aparecer quando trabalhamos com codificações mais específicas, como problema exemplo UTF-8. Pelo fato de usarmos apenas bytes isso pode nos trazer problemas de conversões, aparecendo naturalmente caracteres estranhos onde deveriam ser acentos. Dado este fato aparece a necessidade de estudar outros tipos de “escritores” e “leitores” como são os Readers e Writers em Java.

Lembre-se que o InputStream, assim como o OutputStream, trabalha com bytes sem se importar com conversões de codificações, mas quando estamos trabalhando com texto puro e precisamos nos preocupar com acentuações e outras codificações específicas, então é importante ficar atento ao seu uso.