**Parte SETE**

**Java I/O**

**Capítulo VINTE E TRÊS**

**Fundamentos de Java I/O**

**Objetivos do Exame**

* Ler e escrever dados a partir do console.
* Usar BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream,  
  FileOutputStream, ObjectOutputStream, ObjectInputStream e PrintWriter do pacote java.io.

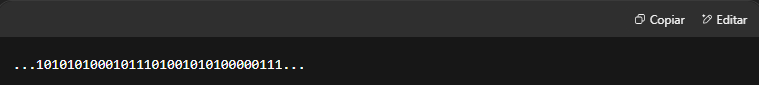
**Fluxos de Entrada/Saída (I/O)**

Java I/O (Entrada/Saída ou Leitura/Escrita) é um tópico extenso, mas vamos começar definindo  
o que é um fluxo (stream) em I/O.

Embora conceitualmente sejam meio parecidos, os fluxos de I/O não estão relacionados de forma alguma  
com a API Stream. Portanto, todas as referências a fluxos neste capítulo se referem a fluxos de I/O.

Em palavras simples, um fluxo é uma sequência de dados.

No contexto deste capítulo, essa sequência de dados é o conteúdo de um arquivo. Tome como exemplo esta sequência de bytes:



Quando lemos essa sequência de bytes de um arquivo, estamos lendo um fluxo de entrada.

Quando escrevemos essa sequência de bytes para um arquivo, estamos escrevendo um fluxo de saída.

Além disso, o conteúdo de um arquivo pode ser tão grande que pode não caber na memória, então ao trabalhar com fluxos,  
não podemos focar no fluxo inteiro de uma vez, mas apenas em pequenas porções (seja byte por byte ou um grupo de bytes)  
por vez.

Mas o Java não suporta apenas fluxos de bytes.

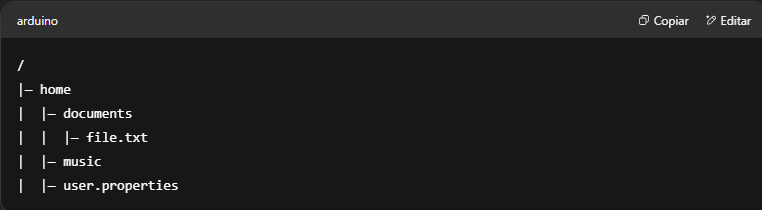
**Arquivos (Files)**

Vamos pensar em um arquivo como um recurso que armazena dados (seja no formato de bytes ou de caracteres).

Arquivos são organizados em diretórios, os quais, além de arquivos, podem conter outros diretórios também.

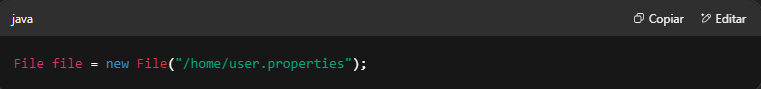
Arquivos e diretórios são gerenciados por um sistema de arquivos.

Sistemas operacionais diferentes podem usar sistemas de arquivos diferentes, mas os arquivos e diretórios são organizados de forma hierárquica. Por exemplo, em um sistema de arquivos baseado em Unix, isso seria:



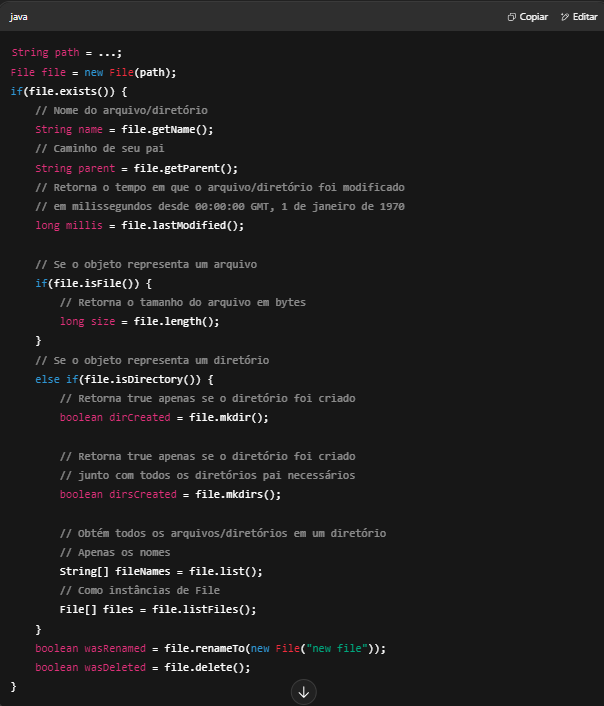
Onde um arquivo ou um diretório pode ser representado por uma String chamada **path (caminho)**, onde o valor à esquerda de um caractere separador (que muda entre sistemas de arquivos) é o pai do valor à direita do separador, como /home/documents/file.txt, ou c:\home\documents\file.txt no Windows.

Em Java, a classe java.io.File representa tanto um arquivo quanto o caminho de um diretório de um sistema de arquivos (não existe uma classe Directory, ao menos na API padrão de I/O do Java, para diretórios):



Quando você cria uma instância da classe File, você **não está criando um novo arquivo**, você está apenas criando um objeto que pode representar um arquivo ou diretório real (ele pode nem mesmo existir ainda).

Mas uma vez que você tenha um objeto File, pode usar alguns métodos para trabalhar com esse arquivo/diretório. Por exemplo:



**Compreendendo o pacote java.io**

Agora que revisamos o básico, podemos dissecar a API de I/O do Java.

Há muitas classes no pacote java.io, mas neste capítulo, revisaremos apenas aquelas cobradas no exame.

Podemos começar listando quatro **CLASSES ABSTRATAS** que são as classes-pai de todas as outras.

As duas primeiras lidam com fluxos de bytes:

* InputStream
* OutputStream

As outras duas lidam com fluxos de caracteres:

* Reader
* Writer

Assim, podemos dizer que todas as classes que têm **Stream** no nome leem ou escrevem fluxos de **BYTES**.

E todas as classes que têm **Reader** ou **Writer** no nome leem ou escrevem fluxos de **CARACTERES**.

Você não deve ter problema em saber que classes que têm **Input** ou **Reader** no nome são para **LEITURA** (seja de bytes ou caracteres).

Que todas as classes que têm **Output** e **Writer** no nome são para **ESCRITA** (seja de bytes ou caracteres).

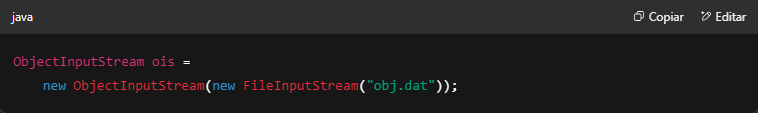
E que toda classe que **LÊ** algo (ou faz isso de certa forma), tem uma classe correspondente que **ESCREVE** aquele algo (ou faz isso daquela certa forma), como FileReader e FileWriter.

No entanto, essa última regra tem exceções. As que você deve conhecer são PrintStream e PrintWriter (olhando para o nome acho que é óbvio que essas classes são apenas para saída, mas ao menos, você pode dizer qual trabalha com bytes e qual com caracteres).

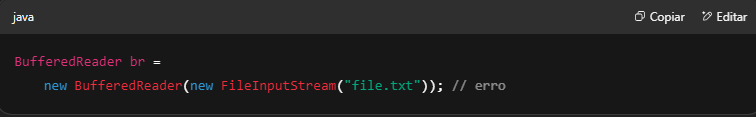
A seguir, temos classes que têm **Buffered** no nome, que usam um buffer para ler ou escrever dados em grupos (de bytes ou caracteres) para fazer isso de forma mais eficiente.

Por fim, com base na ideia de que podemos usar algumas classes em **COMBINAÇÃO**, podemos classificar ainda mais as classes da API como **embrulhadoras (wrappers)** e **não-embrulhadoras (non-wrappers)**.

Classes **não-embrulhadoras** geralmente recebem uma instância de File ou uma String para criar uma instância, enquanto classes **embrulhadoras** recebem outra classe de fluxo para criar uma instância. O seguinte é um exemplo de uma classe embrulhadora:



Ao combinar classes dessa forma, em quase todos os casos, **não é válido** misturar conceitos **OPORTOS**, como combinar um Reader com um Writer ou um Reader com um InputStream:

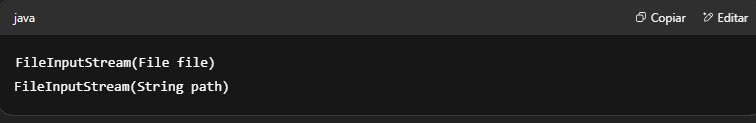


Essa classificação não é tão evidente ao olhar apenas para o nome das classes, mas se você souber o que cada uma delas faz e pensar a respeito, saberá se elas podem ou não embrulhar outras classes do java.io.

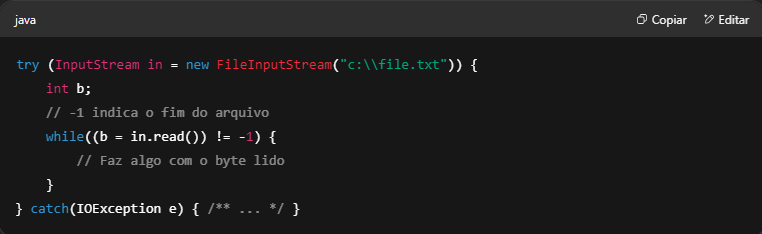
**FileInputStream**

FileInputStream lê bytes de um arquivo. Ela herda de InputStream.

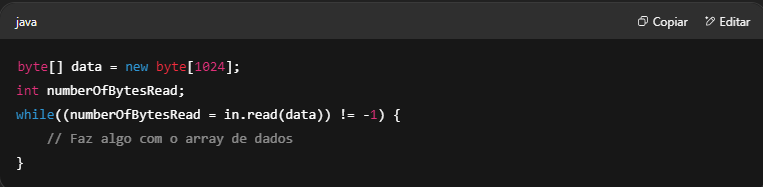
Pode ser criada tanto com um objeto File quanto com um caminho String:



Veja como usá-la:



Há também um método read() que lê bytes em um array de bytes:



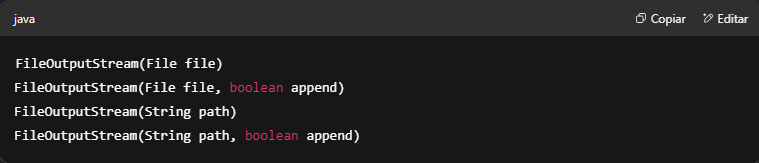
Todas as classes que revisaremos devem ser fechadas. Felizmente, elas implementam AutoCloseable, então podem ser usadas com try-with-resources.

Além disso, quase todos os métodos dessas classes lançam IOExceptions ou uma de suas subclasses (como FileNotFoundException, que é bastante descritiva).

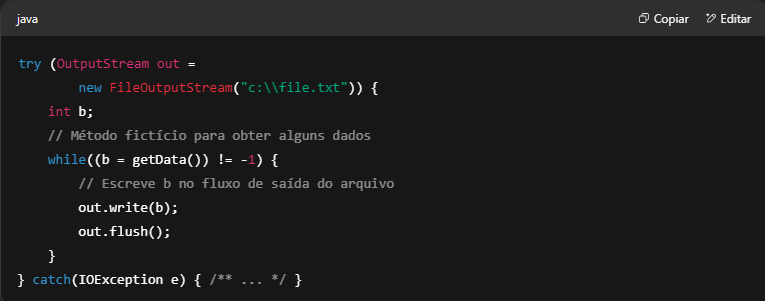
**FileOutputStream**

FileOutputStream grava bytes em um arquivo. Ela herda de OutputStream.

Pode ser criada com um objeto File ou com um caminho String, e um boolean opcional que indica se você quer sobrescrever ou **acrescentar** ao arquivo se ele já existir (por padrão, ele é sobrescrito):



Veja como usá-la:



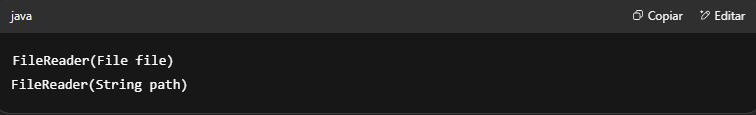
Quando você escreve em um OutputStream, os dados podem ser armazenados internamente em cache na memória e escritos no disco mais tarde.  
Se você quiser ter certeza de que todos os dados foram escritos no disco **sem precisar fechar** o OutputStream, pode chamar o método flush() de tempos em tempos.

FileOutputStream também contém versões sobrecarregadas de write() que permitem escrever dados contidos em um array de bytes.

**FileReader**

FileReader lê caracteres de um arquivo de texto. Ela herda de Reader.

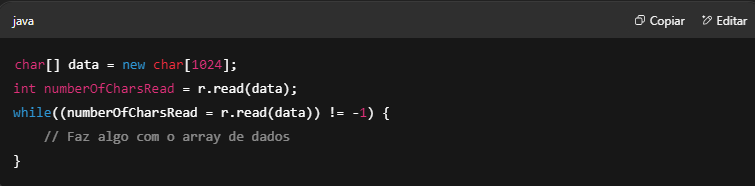
Pode ser criada com um objeto File ou com um caminho String:



Veja como usá-la:



Também há um método read() que lê caracteres em um array de chars:

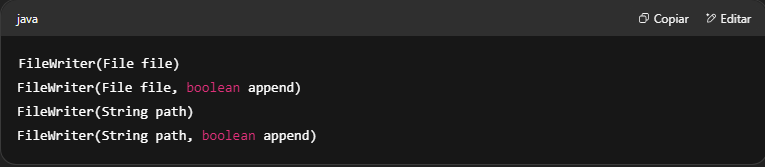


FileReader assume que você deseja decodificar os caracteres no arquivo usando  
a **codificação de caracteres padrão** da máquina na qual seu programa está sendo executado.

**FileWriter**

FileWriter grava caracteres em um arquivo de texto. Ela herda de Writer.

Pode ser criada com um objeto File ou com um caminho String, e um boolean opcional  
que indica se você deseja sobrescrever ou **acrescentar** ao arquivo, caso ele já exista  
(por padrão, ele é sobrescrito):



Veja como usá-la:



Assim como um OutputStream, os dados podem ser armazenados internamente em cache na memória  
e escritos no disco posteriormente. Se você quiser garantir que todos os dados foram realmente  
gravados no disco sem precisar fechar o FileWriter, pode chamar o método flush() de vez em quando.

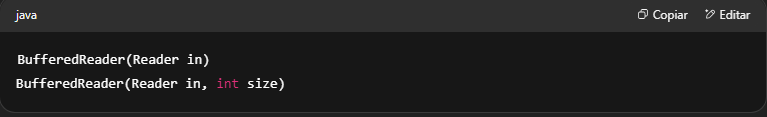
FileWriter também contém versões sobrecarregadas de write() que permitem escrever dados  
contidos em um array de char, ou em uma String.

FileWriter assume que você deseja codificar os caracteres no arquivo usando  
a **codificação de caracteres padrão** da máquina em que seu programa está sendo executado.

**BufferedReader**

BufferedReader lê texto de um fluxo de caracteres. Em vez de ler um caractere por vez,  
BufferedReader lê um bloco grande de uma vez para dentro de um buffer. Ela herda de Reader.

Esta é uma **classe embrulhadora** que é criada passando-se um Reader para seu construtor e, opcionalmente, o tamanho do buffer:



BufferedReader possui um método extra de leitura (além dos herdados de Reader), readLine().  
Veja como usá-lo:



Quando o BufferedReader é fechado, ele também fechará a instância de Reader da qual lê.

**BufferedWriter**

BufferedWriter grava texto em um fluxo de caracteres, armazenando caracteres em buffer para maior eficiência.  
Ela herda de Writer.

Esta é uma **classe embrulhadora**, que é criada passando-se um Writer para seu construtor e, opcionalmente, o tamanho do buffer:



BufferedWriter possui um método extra de escrita (além dos herdados de Writer), newLine().  
Veja como usá-lo:



Como os dados são escritos primeiro em um buffer, você pode chamar o método flush() para garantir  
que o texto escrito até aquele momento seja realmente gravado no disco.

Quando o BufferedWriter é fechado, ele também fechará a instância de Writer para a qual escreve.

**ObjectInputStream / ObjectOutputStream**

O processo de converter um objeto para um formato de dados que pode ser armazenado (em um arquivo, por exemplo)  
é chamado de **serialização**, e converter esse formato de dados armazenado de volta em um objeto é chamado de **desserialização**.

Se você quiser serializar um objeto, sua classe deve implementar a interface java.io.Serializable,  
a qual **não possui métodos para implementar**, ela apenas **marca** os objetos daquela classe como serializáveis.

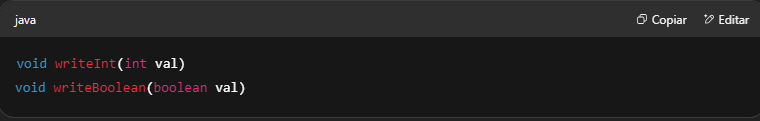
Se você tentar serializar uma classe que **não** implementa essa interface, uma  
java.io.NotSerializableException (uma subclasse de IOException) será lançada em tempo de execução.

ObjectOutputStream permite que você serialize objetos para um OutputStream, enquanto  
ObjectInputStream permite que você desserialize objetos a partir de um InputStream. Portanto, ambas são consideradas **classes embrulhadoras (wrapper classes)**.

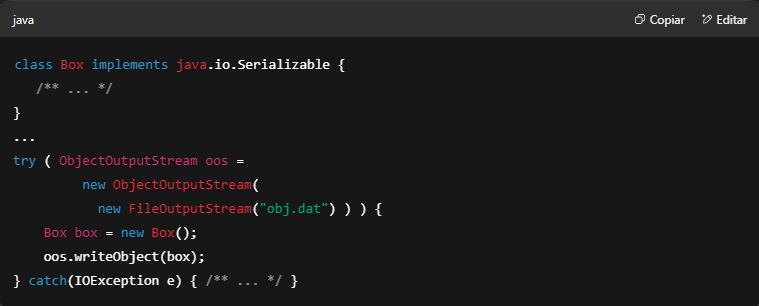
Aqui está o construtor da classe ObjectOutputStream:



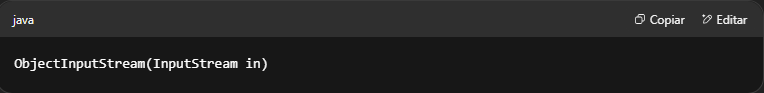
Essa classe possui métodos para gravar muitos tipos primitivos, como:



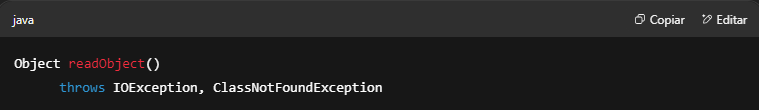
Mas o mais útil é writeObject(Object). Aqui está um exemplo:



Para desserializar o arquivo obj.dat, usamos a classe ObjectInputStream. Aqui está seu construtor:



Essa classe possui métodos para ler muitos tipos de dados, entre eles:



Observe que ele retorna um tipo Object. Assim, temos que **fazer o cast explicitamente**.  
Isso pode levar a uma ClassCastException em tempo de execução. Note também que esse método lança  
uma ClassNotFoundException (uma exceção verificada), caso a classe de um objeto serializado **não possa ser encontrada**.

Aqui está um exemplo:

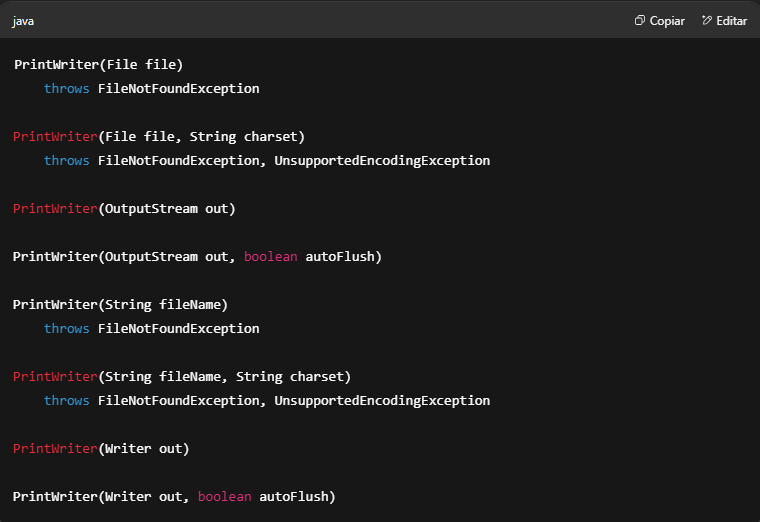


Duas observações importantes:

1. Ao desserializar um objeto, o construtor e quaisquer blocos de inicialização **não são executados**.
2. Objetos null **não são serializados/desserializados**.

**PrintWriter**

PrintWriter é uma subclasse de Writer que grava dados formatados em outro fluxo (embrulhado),  
até mesmo um OutputStream. Veja seus construtores:



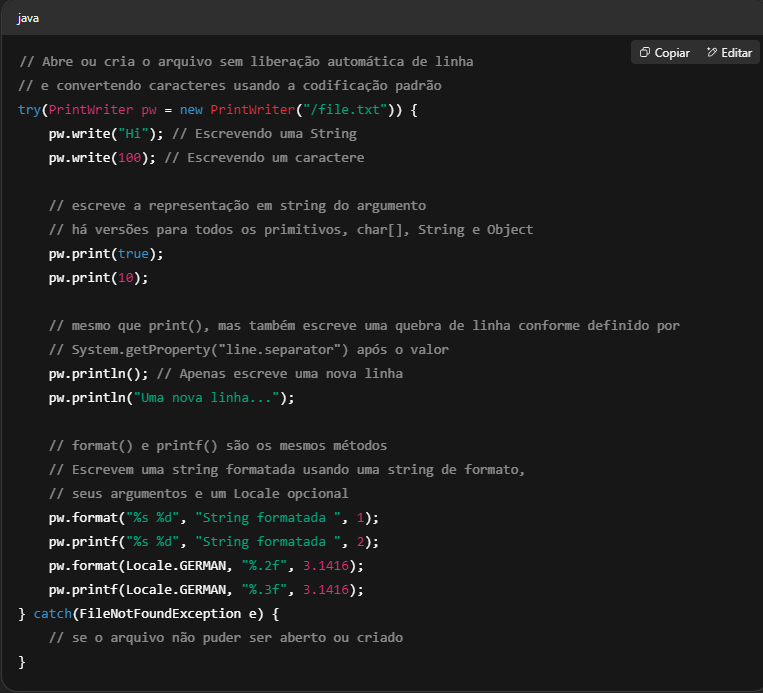
Por padrão, ela usa o conjunto de caracteres (charset) **padrão da máquina** em que o programa está sendo executado,  
mas ao menos essa classe aceita os seguintes charsets (há outros opcionais):

* US-ASCII
* ISO-8859-1
* UTF-8
* UTF-16BE
* UTF-16LE
* UTF-16

Como qualquer Writer, essa classe possui o método write() que vimos em outras subclasses de Writer,  
mas ela os sobrescreve para **não lançar uma IOException**.

Ela também adiciona os métodos format(), print(), printf(), println().

Veja como usar essa classe:



Você pode aprender mais sobre strings de formato para format() e printf() em:  
<https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html>

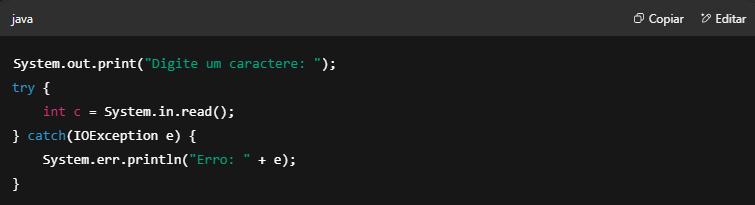
**Fluxos padrão (Standard streams)**

O Java inicializa e fornece três objetos de fluxo como campos public static da classe java.lang.System:

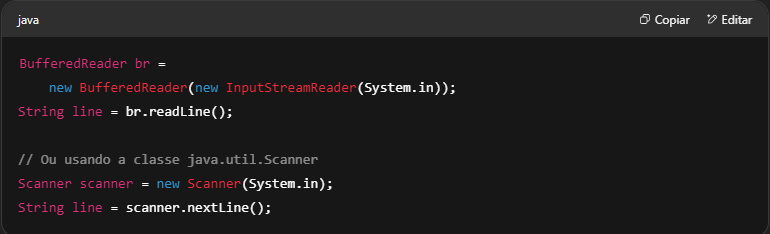
* **InputStream System.in**  
  O fluxo de entrada padrão (tipicamente a entrada do teclado)
* **PrintStream System.out**  
  O fluxo de saída padrão (tipicamente a saída de exibição padrão)
* **PrintStream System.err**  
  O fluxo de saída de erro padrão (tipicamente a exibição de erros)

PrintStream faz exatamente as mesmas coisas e tem os mesmos recursos de PrintWriter,  
só que trabalha apenas com OutputStreams.

O exemplo a seguir mostra como ler um único caractere (um byte) do terminal:



Ou para ler Strings:



**java.io.Console**

Desde o Java 6, temos a classe java.io.Console para acessar o console da máquina  
na qual seu programa está sendo executado.

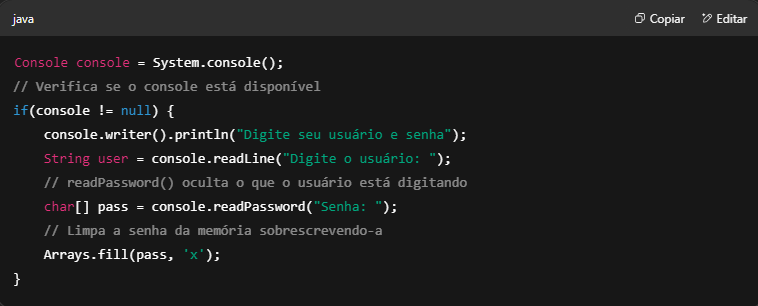
Você pode obter uma referência a essa classe (ela é um singleton) com System.console().

Mas tenha em mente que, se seu programa estiver sendo executado em um ambiente que **não tem acesso a um console**  
(como uma IDE ou se seu programa estiver rodando como um processo em segundo plano),  
System.console() retornará null.

Com o objeto Console, você pode facilmente ler a entrada do usuário com o método readLine()  
e até mesmo ler senhas com readPassword().

Para saída, essa classe possui os métodos format() e printf() que funcionam exatamente como os do PrintWriter.

Finalmente, os métodos reader() e writer() retornam uma instância de Reader e Writer, respectivamente:



readPassword() retorna um array de char para que ele possa ser totalmente e imediatamente removido da memória  
(Strings vivem em um pool de memória e são coletadas por garbage collector, então um array é mais seguro).

**Pontos-chave**

* Um **fluxo de I/O** é uma sequência de dados que representa o conteúdo de um arquivo.
* Um **fluxo de entrada** é para leitura e um **fluxo de saída** é para escrita.
* No pacote java.io, podemos encontrar classes para trabalhar com **fluxos de bytes e de caracteres**.
* Existem quatro classes **abstratas principais** das quais as demais classes derivam:  
  InputStream, OutputStream, Reader, Writer.
* As classes java.io podem ser classificadas como:
  + Para fluxos de **bytes** ou **caracteres**
  + Para **entrada** ou **saída**
  + **Embrulhadoras (wrappers)** ou **não-embrulhadoras (non-wrappers)**

O Java inicializa e fornece três objetos de fluxo como campos public static da classe java.lang.System:

* InputStream System.in  
  O fluxo de entrada padrão (normalmente o teclado)
* PrintStream System.out  
  O fluxo de saída padrão (normalmente a saída da tela)
* PrintStream System.err  
  O fluxo de erro padrão (normalmente a exibição de erros)

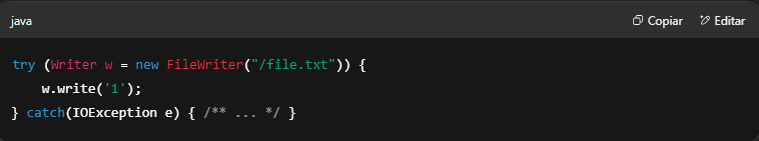
**A tabela a seguir resume as classes revisadas neste capítulo:**

| **Classe** | **Estende de** | **Argumentos principais do construtor** | **Métodos principais** | **Descrição** |
| --- | --- | --- | --- | --- |
| File | — | String | exists, getParent, isDirectory, isFile, listFiles, mkdirs, delete, renameTo | Representa um arquivo ou diretório. |
| FileInputStream | InputStream | File, String | read | Lê conteúdo de arquivo como bytes. |
| FileOutputStream | OutputStream | File, File + boolean, String, String + boolean | write | Escreve conteúdo de arquivo como bytes. |
| FileReader | Reader | File, String | read | Lê conteúdo de arquivo como caracteres. |
| FileWriter | Writer | File, File + boolean, String, String + boolean | write | Escreve conteúdo de arquivo como caracteres. |
| BufferedReader | Reader | Reader, Reader + int | readLine, read | Lê texto em fluxo de caracteres, com buffer. |
| BufferedWriter | Writer | Writer, Writer + int | newLine, write | Escreve texto em fluxo de caracteres, com buffer. |
| ObjectInputStream | InputStream | InputStream | readObject | Desserializa dados primitivos e objetos. |
| ObjectOutputStream | OutputStream | OutputStream | writeObject | Serializa dados primitivos e objetos. |
| PrintWriter | Writer | File, OutputStream, String, Writer | format, print, printf, println | Escreve dados formatados em um fluxo de caracteres. |
| Console | — | — | readLine, readPassword, format, printf | Acessa o console, se houver. |

**Autoavaliação**

**1. Qual das seguintes é uma maneira válida de criar um objeto PrintWriter?**  
A. new PrintWriter(new Writer("file.txt"));  
B. new PrintWriter();  
C. new PrintWriter(new FileReader("file.txt"));  
D. new PrintWriter(new OutputStream("file.txt"));

**2. Dado:**



Qual das alternativas a seguir é o resultado da execução das linhas acima se o arquivo já existir?  
A. Ele sobrescreve o arquivo  
B. Ele acrescenta 1 ao arquivo  
C. Nada acontece, já que o arquivo já existe  
D. Uma IOException é lançada

**3. Qual das seguintes é o tipo do objeto System.in?**  
A. Reader  
B. InputStream  
C. BufferedReader  
D. BufferedInputStream

**4. Dado:**



Qual das alternativas a seguir é o resultado da execução das linhas acima?  
A. Nada é impresso, a classe é serializada em d.dat  
B. Nada é impresso, mas a classe não é serializada  
C. Erro  
D. Uma exceção em tempo de execução é lançada

**5. O que o método flush() faz?**  
A. Marca o fluxo como pronto para ser escrito  
B. Fecha o fluxo  
C. Escreve os dados armazenados em disco em um cache  
D. Escreve os dados armazenados em um cache no disco