Usando I/O

Sistema I/O de Java

- O sistema de I/O Java é baseado na hierarquia de classe, não foi possível apresentar sua teoria e detalhes sem antes discutir as classes, a herança e as exceções.
- O sistema de I/O Java é bem grande, contendo muitas classes, interfaces e métodos. Parte da razão de seu tamanho é que Java define dois sistemas de I/O completos: um para I/O de bytes e outro para I/O de caracteres.
- Serão apresentados os recursos mais usados e importantes. Felizmente os sistema de I/O Java é coeso e coerente; uma vez que você entenda os aspectos básicos, o resto será fácil de dominar.

I/O Java é baseado em fluxos

- Os programas Java executam I/O por intermédio de fluxos. Uma fluxo é uma abstração que produz ou consome informações. Ele é vinculado a um dispositivo físico pelo sistema I/O de Java.
- Todos os fluxos se comportam igualmente, mesmo que os dispositivos físicos aos quais estejam vinculados sejam diferentes.
- Os mesmos métodos usados para a gravação no console também podem ser usados na gravação em um arquivo em disco.

Fluxos de bytes e Fluxos de caracteres

- Versões modernas de Java definem dois tipos de fluxos: de bytes e de caracteres.
- Os fluxos de bytes fornecem um meio conveniente para o tratamento de entrada e saída de bytes. Eles são usados, por exemplo, na leitura ou gravação de dados binários. São especialmente úteis no trabalho com arquivos.
- Os fluxos de caracteres foram projetados para o tratamento da entrada e saída de caracteres. Eles usam o Unicode e, portanto podem ser internacionalizados.
- Em alguns casos, os fluxos de caracteres são mais eficientes do que os fluxos de bytes.

Fluxos de bytes e Fluxos de caracteres

- O fato de Java definir dois tipos de fluxos diferentes aumenta e muito o sistema de I/O, porque dois conjuntos de hierarquias de classes separados são necessários.

- O grande número de classes pode fazer o sistema de I/O parecer mais assustados do que realmente é.

- No nível mais baixo, todo o I/O continua orientado a bytes. Os fluxos baseados em caracteres apenas fornecem um meio conveniente e eficiente de tratamento de caracteres.

Classes de fluxos de bytes

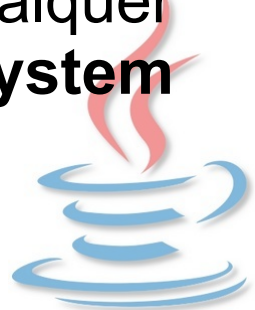
- Os fluxos de bytes são definidos com uso de duas hierarquias de classes. No topo delas estão duas classes abstratas: **InputStream** e **OutputStream**.
- **InputStream** define as características comuns a fluxos de entrada de bytes e **OutputStream** descreve o comportamento dos fluxos de saída de bytes.
- A partir do **InputStream** e **OutputStream**, são criadas muitas subclasses concretas que oferecem funcionalidade variada e tratam os detalhes de leitura e gravação em vários dispositivos, como arquivos em disco.

Classes de fluxos de caracteres

- Os fluxos de caracteres são definidos com uso de duas hierarquias de classes encabeçadas pelas seguintes duas classes abstratas: **Reader** e **Writer**.
- **Reader** é usada para entrada e **Writer** para saída. As classes concretas derivadas de **Reader** e **Writer** operam com fluxos de caracteres Unicode.
- De **Reader** e **Writer** são derivadas muitas subclasses concretas que tratam várias situações de I/O.
- Em geral, as classes baseadas em caracteres são equivalentes às classes baseadas em bytes.

Fluxos Predefinidos

- Todos os programas Java importam automaticamente o pacote **java.lang**.
- Esse pacote define uma classe chamada **System**, que encapsula vários aspectos do ambiente de tempo de execução.
- Entre outras coisas, ela contém três variáveis de fluxos predefinidas, chamadas **in**, **out** e **err**.
- Esses campos são declarados como **public, final** e **static** dentro de **System**, ou seja, podem ser usados por qualquer parte do programa e sem referência a um objeto **System** específico.

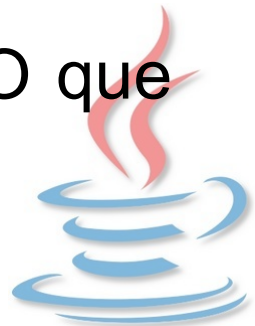


Fluxos Predefinidos

- **System.out** é o fluxo de saída básico ; por padrão, ele usa o console. **System.in** é a entrada básica, que por padrão é o teclado. **System.err** é o fluxo de erro básico, que por padrão também usa o console.
- No entanto, esses fluxos podem ser redirecionados para qualquer dispositivo de I/O compatível.
- **System.in** é um objeto de tipo **InputStream, System.out** e **System.err** são objetos de tipo **PrintStream.** Eles são fluxos de bytes, mesmo que normalmente sejam usados na leitura e gravação de caracteres no console.

Usando fluxos de bytes

- Em geral os métodos de **InputStream** e **OutputStream** podem lançar uma **IOException** em caso de erro.
- Os métodos definidos por essas dias classes abstratas estão disponíveis para todas as suas subclasses.
- Logo, formam um conjunto mínimo de funções de I/O que todos os fluxos de bytes terão.



Lendo a entrada do console

- Originalmente, a única maneira de ler entradas de console era usar um fluxo de bytes e muitos códigos Java ainda usam somente fluxos de bytes.
- Atualmente, você pode usar fluxos de bytes ou caracteres.
- Para códigos comerciais, o método preferido de leitura de entradas no console é com um fluxo orientado a caracteres.
- Isso facilita a internacionalização e a manutenção do programa.
- Exemplo da pg. 331, mostra leitura pelo teclado.

Gravando a saída do console

- Como no caso da entrada do console, originalmente Java só fornecia fluxos de bytes para a saída do console.
- A saída do console é obtida mais facilmente com os métodos **print()** e **println()**. Esses métodos são definidos pela classe **PrintStream**. Mesmo com o **System.out** sendo um fluxo de bytes, é aceitável usar esse fluxo para saídas simples no console.
- Já que **PrintStream** é um fluxo de saída de derivado de **OutputStream**, ele também implementa o método de baixo nível **write()**. Portanto, é possível gravar no console usando **write()**.

- Veremos no Exemplo da pg. 332

Lendo e gravando arquivos usando fluxos de bytes

- Java fornece várias classes e métodos que permitem a leitura e gravação de arquivos.
- É claro que os tipos de arquivos mais comuns são os em disco.
- Em Java, todos os arquivos são orientados a bytes e a linguagem fornece métodos para a leitura e gravação de bytes em um arquivo.
- Logo, é muito comum ler e gravar arquivos usando fluxos de bytes. No entanto, Java permite o encapsulamento de um fluxo de arquivo orientado a bytes dentro de um objeto baseado em caracteres.

Lendo e gravando arquivos usando fluxos de bytes

- Para criar um fluxo de bytes vinculados a um arquivo, use **FileInputStream** ou **FileOutputStream**.
- Para abrir um arquivo, simplesmente crie um objeto de uma dessas classes, especificando o nome do arquivo como argumento do construtor.
- Uma vez que o arquivo for aberto, você poderá ler e gravar nele.

Gerando entradas em um arquivo

- Um arquivo é aberto para gerar entradas com a criação de um objeto FileInputStream. O construtor abaixo é muito usado:

  FileInputStream(String *nomeArquivo*) trows FileNotFoundException

- Aqui, *nomeArquivo* especifica o nome do arquivo que você deseja abrir. Se ele não existir, uma **FileNotFoundException** será lançada.

- **FileNotFoundException** é uma subclasse de **IOException**.

- Para ler em um arquivo, você pode usar **read()**. A versão que usaremos é mostrada a seguir:

  int read() throws **IOException**

Gerando entradas em um arquivo

- Sempre que é chamado, **read()** lê um único byte no arquivo e o retorna como valor inteiro.
- Ele retorna -1 quando o fim do arquivo é alcançado e lança uma **IOException** quando ocorre um erro.
- Portanto, essa versão de **read()** é igual a usada na leitura a partir do console.
- Quando tiver terminado de usar um arquivo, você deve fechá-lo chamando o método **close()**.

Gerando entradas em um arquivo

- O fechamento de um arquivo libera os recursos do sistema alocados para ele.
- Permitindo que seja, usados por outros arquivos. Não fechar um arquivo pode resistir em “vazamento de memória”, porque recursos não usados permanecem alocados.
- As vezes, é mais fácil encapsular as partes de um programa referentes à abertura e ao acesso do arquivo dentro do mesmo bloco **try** e então usar um bloco **finally** para fechar o arquivo.
- Os exemplos das pgs. 334 e 335 mostram de maneira prática.

Gravando em um arquivo

- Para abrir um arquivo para saída, crie um objeto FileOutputStream. Aqui são dois construtores normalmente utilizados:

FileOutputStream(String *nomeArquivo*) trows FileNotFoundException
FileOutputStream(String *nomeArquivo*,boolean *incluir*)
trows FileNotFoundException

- Se o arquivo não puder ser criado, uma FileNotFoundException será lançada.
- Na primeira forma, quando um arquivo de saída é aberto, qualquer arquivo preexistente com o mesmo nome é destruído. Na segunda forma, se incluir for igual a **true**, a saída será acrescida ao fim do arquivo. Caso contrário, o arquivo será sobreposto.

Gravando em um arquivo

- Para gravar em um arquivo, você usará o método **write()**.Sua forma mais simples é mostrada aqui:

void write(int *valbyte*) throws IOException

- Esse método grava o byte especificado por valbyte no arquivo. Embora valbyte seja declarada como um inteiro, só os 8 bits de ordem inferior são gravados no arquivo. Se um erro ocorrer durante a gravação, uma **IOException** será lançada.
- Uma vez que você tiver terminado de usar um arquivo de saída, deve fechá-lo usando o método **close().**
- Exemplo da pg. 338

Fechando automaticamente um arquivo

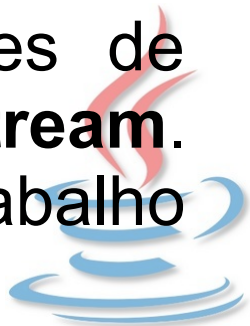
- **Close()**, é assim que os arquivos têm sido fechados desde que Java foi criada.
- Como resultado, essa abordagem está disseminada nos códigos existentes.
- Além disso, ela ainda é válida e útil, porém, JDK 7 adiciona um novo recurso que oferece uma maneira mais otimizada de gerenciar recursos, como os fluxos de arquivos, automatizando o processo de fechamento.
- Ela se baseia em uma nova versão da instrução try chamada try-with resources,e que também é conhecida como gerenciamento automático de recursos.

Fechando automaticamente um arquivo

- A principal vantagem do **try**-with-resources é a de ele impedir a ocorrência de situações em que um arquivo (ou outro recurso) não é liberado quando não é mais necessário.
- Quando o bloco **try** termina, o recurso é liberado automaticamente, ou seja, no caso de um arquivo, ele é fechado automaticamente.
- Uma instrução **try**-with- resources também pode incluir cláusulas **catch** e **finally**.

Fechando automaticamente um arquivo

- A instrução **try**-with-resources só pode ser usada com os recursos que implementam a interface **AutoCloseable** definida por **java.lang**.
- Essa interface que foi adicionada por JDK 7, define o método **close()**.
- **AutoCloseable** é herdada pela interface **Closeable** definida por **java.io**.
- Ambas interfaces são imlementadas pelas classes de fluxo, inclusive **FileInputStream** e **FileOutputStream**. Portanto **try**-with-resources pode ser usada no trabalho com fluxos, o que inclui os fluxos de arquivo.



- Exemplo da pg. 339.