UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Leitura e Escrita de Arquivos Binários usando Java

Programação Orientada a Objetos

Prof. Fabrício M. Lopes
fabricio@utfpr.edu.br

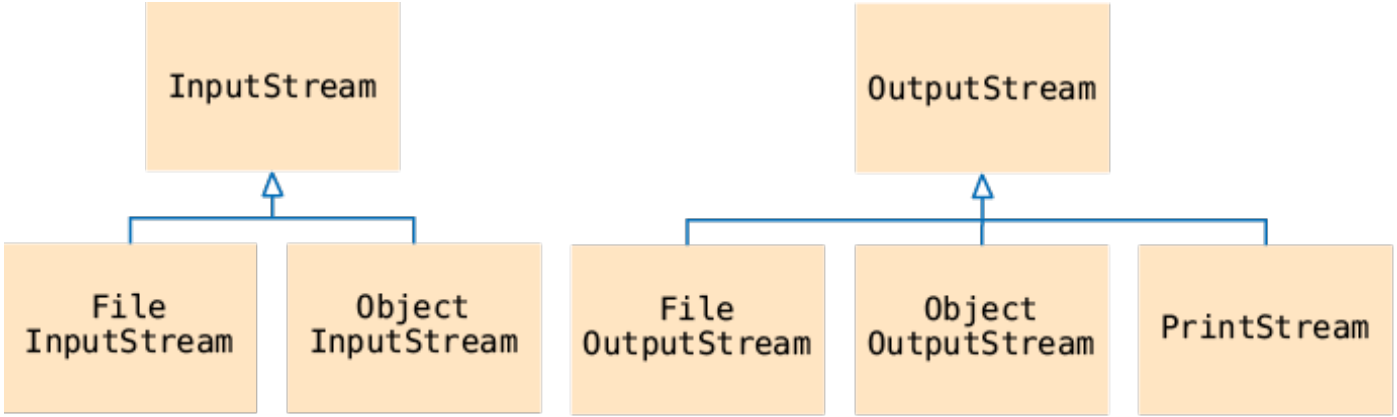
Arquivos Texto e Binários

- Duas formas de armazenar dados: ***texto*** e ***formato binário***.
- Formato de texto: forma legível, como uma sequência de caracteres.
  - Por exemplo, número inteiro 12.345 armazenado como caracteres '1' '2' '3' '4' '5'.
  - Mais conveniente: mais fácil de produzir entrada e de verificar saída.
  - ***Readers*** e ***writers*** são usados para leitura e escrita de textos.

Arquivos Texto e Binários

- ***Formato binário***: os dados são representados em ***bytes***.
- Um byte é composto por 8 bits e podem representar até 256 valores ($256 = 2^8$).
  - Por exemplo, número inteiro 12.345 armazenado como sequência de quatro bytes 0 0 48 57
    - 12.345 = 48 x 256 + 57.
  - Mais compacto e mais eficiente.
- Os ***Streams*** lidam com os dados binários.

Arquivos Binários



O Java fornece dois conjuntos de classes para manipulação de entrada e saída. Os fluxos de entrada e saída tratam de dados binários. A Figura acima exibe uma parte da hierarquia das classes Java para entrada e saída em formato binário.

Escrita e Leitura em Binário

- Para ler dados binários de um arquivo em disco, o objeto usado é o **FileInputStream**:
  - *InputStream entrada = new FileInputStream("input.bin");*
- De forma similar, o objeto **FileOutputStream** é usado para gravar dados em um arquivo de disco em formato binário:
  - *OutputStream saida = new FileOutputStream("output.bin");*

InputStream e FileInputStream

- A classe **InputStream** tem o método ***read()*** para ler um único byte de cada vez.
- A classe ***FileInputStream*** sobrescreve o método ***read()*** para ler os bytes de um arquivo em disco.
- O método ***InputStream.read()*** retorna um int, não um byte, para que ele possa sinalizar que um byte foi lido ou que o fim da leitura foi alcançado. Ele retorna o byte lido como um número inteiro entre 0 e 255 ou, quando está no final da entrada, retorna -1.

```
InputStream in = new FileInputStream("input.bin");
int next = in.read();
if (next != -1)
{
   Faça algo com o valor da variável next (que deve ser um valor entre 0 e 255)
}
```

InputStream - Exemplo

```
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
/**
 * @author fabricio@utfpr.edu.br
 */
public class InputStreamExemplo {
    public static void main(String[] args) throws IOException {
        URL locator = new URL("http://paginapessoal.utfpr.edu.br/fabricio/imagens/utfpr.jpg");
        InputStream in = locator.openStream();
        int next = in.read();
        while (next != -1) {
            System.out.println(next);
            next = in.read();
        }
        in.close();
    }
}
```

OutputStream e FileOutputStream

- A classe ***OutputStream*** define o método abstrato ***write(int)*** para escrever um único byte, o qual aceita como parâmetro um ***int***.
- A classe ***FileOutputStream*** implementa o método ***write (int)*** para escrever um byte em um arquivo no disco.

```
OutputStream out = new FileOutputStream (“saida.bin”);
int valor = 128; //deve ser um valor entre 0 e 255
out.write(valor);
```

FileOutputStream - Exemplo

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
/**
 * @author fabricio@utfpr.edu.br
 */
public class OutputStreamExemplo {
    public static void main(String[] args) throws IOException {
        URL locator = new URL("http://paginapessoal.utfpr.edu.br/fabricio/imagens/utfpr.jpg");
        InputStream in = locator.openStream();
        FileOutputStream out = new FileOutputStream("/home/fabricio/temp/img.jpg");
        int next = in.read();
        while (next != -1) {
            out.write(next);
            next = in.read();
        }
        in.close();
        out.close();
    }
}
```

Leitura e Escrita em Binário

- Estes métodos básicos ***InputStream.read()*** e ***OutputStream.write()*** são os únicos métodos de entrada e saída que as classes de fluxo de entrada e saída fornecem.
- O pacote de entrada/saída Java é construído com base no princípio de que cada classe deve ter uma responsabilidade muito focada.
- O trabalho de um fluxo de entrada é obter bytes, não analisá-los. Se for necessário ler números, strings ou outros objetos, é necessário combinar a classe com outras classes;

Fluxos de Objetos

- A classe ***ObjectOutputStream*** pode ser usada para salvar objetos inteiros em arquivo no disco.
- A classe ***ObjectInputStream*** podem ler esses objetos que foram anteriormente gravados.
- Para isso, devem ser utilizados os ***streams***, devido aos objetos serem escritos e lidos em formato binário.

ObjectOutputStream

- Para escrever objetos, não é necessário escrever byte.
- A classe ***ObjectOutputStream*** pode salvar objetos inteiros em disco usando o método ***writeObject()***.
- O método ***writeObject*** recebe como parâmetro um **Object**[1] para a escrita.
  - Por exemplo, é possível escrever um objeto Cliente em um arquivo da seguinte forma:

```
Cliente cli = ...;

ObjectOutputStream saida = new ObjectOutputStream(new FileOutputStream("clientes.dat"));

out.writeObject(cli);
```


[1] O Object é a raiz da hierarquia de classes. Todas as classes têm Object como uma superclasse.

ObjectInputStream

- Após a escrita de objeto, é possível a sua leitura usando o método ***readObject()*** da classe ***ObjectInputStream***.
- Esse método retorna um objeto ***Object***, então é necessário saber *a priori* qual foi a classe do objeto que foi escrito e usar uma conversão de tipos.

```
ObjectInputStream in = new ObjectInputStream( new FileInputStream("clientes.dat"));
Cliente cli = (Cliente) in.readObject();
```

O método `readObject()` pode disparar a exceção `ClassNotFoundException`, a qual deve ser tratada.

Generalizando a escrita de objetos

- É possível escrever vários objetos de uma única vez utilizando uma composição, como vetores ou matrizes, ou seja, um objeto dentro de outro objeto.
- Logo, caso os clientes estejam armazenados em um objeto de outra classe, por exemplo em um ***ArrayList***, então é possível simplesmente salvar e restaurar esse objeto composto.
- Então seu objeto (***ArrayList***), e todos os objetos clientes que ele contém, são automaticamente salvos e restaurados também.
- Esta é uma capacidade importante e recomendada de programação.

```
ArrayList<Cliente> clientes = new ArrayList<>();
// Então é possível incluir muitos objetos da classe Cliente no objeto clientes
saida.writeObject(clientes);

//depois é possível realizar a leitura de uma única vez
ArrayList<Cliente> clientes =  (ArrayList<Cliente>) entrada.readObject();
```

Interface Serializable

- Objetos que são escritos em um fluxo **devem** pertencer a uma classe que implementa a interface **Serializable**.
  - Caso não implemente, é disparada a exceção: ***java.io.NotSerializableException.***
- A interface Serializable não possui métodos, é uma interface de marcação.
- ***Serialização***[1]: processo de salvar objetos em um fluxo, sendo que cada objeto escrito recebe um número serial.
- Na leitura o número serial é verificado para conferir a classe utilizada na gravação.

```
public class Cliente implements java.io.Serializable
{
 . . .
}
```


[1] https://docs.oracle.com/javase/6/docs/platform/serialization/spec/serialTOC.html

serialVersionUID

- Cada classe pode ser versionada de modo a identificar a versão dessa classe que será usada para a escrita e leitura usando o atributo:
  - private static final long ***serialVersionUID*** = 348749589581939 3L;
- O ***serialVersionUID*** é um identificador único[1], composto por um código hash 64 bits considerando o nome da classe, nome da interface, métodos e atributos.
- Se o ***serialVersionUID*** não for declarado para uma classe, o valor padrão é o hash para essa classe.

[1] https://docs.oracle.com/javase/6/docs/platform/serialization/spec/class.html#4100

serialVersionUID

- É fortemente recomendado que todas as classes serializáveis declarem explicitamente os valores ***serialVersionUID***.
- O cálculo padrão do ***serialVersionUID*** é altamente sensível aos detalhes da classe que podem variar dependendo das atualizações nas classes, podendo resultar em conflitos inesperados do serialVersionUID durante a leitura, causando falha.

Apresentação dos Exemplos

- TestaArquivoBinario.java

- Cliente.java

Referências

- DEITEL, P.J. Java - Como Programar. Porto Alegre: Bookman, 2001.
- NIEMEYER, Patrick. Aprendendo java 2 SDK. Rio de Janeiro: Campus, 2000.
- MORGAN, Michael. Java 2 para Programadores Profissionais. Rio de Janeiro: Ciência Moderna, 2000.
- HORSTMANN, Cay, S. e CORNELL, Gary. Core Java 2. São Paulo: Makron Books, 2001 v.1. e v.2.