



Trabalhando com arquivos

Importação e exportação de arquivos

Quando os itens de dados são armazenados em um sistema de computador, eles podem ser armazenados por períodos variáveis - temporária ou permanentemente.

- O armazenamento temporário é geralmente chamado de memória do computador ou memória de acesso aleatório (RAM). Ao escrever um programa Java que armazena um valor em uma variável, você está usando armazenamento temporário; o valor que você armazena é perdido quando o programa termina ou o computador perde energia. Esse tipo de armazenamento é volátil.
- O armazenamento permanente, por outro lado, não é perdido quando um computador perde energia; é não volátil. Ao gravar um programa Java e salvá-lo em um disco, você está usando armazenamento permanente.

Os arquivos existem em dispositivos de armazenamento permanente, como discos rígidos, discos Zip, unidades USB, bobinas ou cassetes de fita magnética e discos compactos. Arquivos de computador são o equivalente eletrônico de arquivos de papel geralmente armazenados em gabinetes de arquivos em escritórios.

Quando você trabalha com arquivos armazenados em um aplicativo normalmente realiza todas ou algumas das seguintes tarefas:



- Determinando se e onde um arquivo existe
- Abrindo um arquivo
- Lendo dados de um arquivo
- Gravando informações em um arquivo
- Fechando um arquivo




Usando a Classe File

Você pode usar a classe File do Java para reunir informações sobre o arquivo, como tamanho, data de modificação mais recente e se o arquivo existe. Você deve incluir a seguinte instrução para usar a classe File:

```
import java.io.File;
```


O pacote **java.io** contém todas as classes usadas no processamento de arquivos, portanto, geralmente é mais fácil importar o pacote inteiro usando o caractere curinga *, da seguinte maneira:

```
import java.io. *;
```

A classe File é uma subclasse da classe Object. Você pode criar um objeto File usando um construtor que inclua um nome de  ,arquivo como argumento, por exemplo, faça a seguinte declaração quando Data.txt for um arquivo na pasta raiz do projeto:

File nomeArquivo = new file (“Data.txt”);

Abaixo está a lista de alguns métodos importantes da classe File com finalidade e assinatura de método

Nome do método / Assinatura	Objetivo 
boolean canRead()	Retorna true se um arquivo estiver legível
boolean canWrite()	Retorna true se um arquivo for gravável
boolean exists()	Retorna true se o arquivo existir
String getName()	Retorna o nome do arquivo
String getPath()	Retorna o caminho do arquivo
String getParent()	Retorna o nome da pasta na qual o arquivo pode ser encontrado
long length()	Retorna o tamanho do arquivo
long lastModified()	Retorna a hora em que o arquivo foi modificado pela última vez; esse tempo depende do sistema e deve ser usado apenas para comparação com os tempos de outros arquivos, não como um tempo absoluto
boolean isDirectory()	Quando você cria um objeto File e ele é um diretório, o método isDirectory () retornará true.

Vamos entender a implementação desse método com a ajuda do programa java. No método main (), um objeto File chamado meuArquivo é declarado. A String passada para o construtor é “SomeData.txt”, que é o nome do sistema do arquivo armazenado. Em outras palavras, embora SomeData.txt possa ser o nome de um arquivo armazenado quando o sistema operacional se refere a ele, o arquivo é conhecido como meuArquivo no aplicativo. Precisamos criar o arquivo Data.txt no diretório

raiz do projeto, caso contrário, receberemos uma mensagem dizendo “O arquivo não existe”.



```
Import java.io.File;
```

```
public class MetodosClasseArquivo{
```

```
    public static void main(String[] args){
```

```
        File meuArquivo=new File("Data.txt");
```

```
        if(meuArquivo.exists()){
```

```
            System.out.println(meuArquivo.getName()+"  
existe");
```

```
            System.out.println("O arquivo tem  
"+meuArquivo.length()+" bytes");
```

```
            if(meuArquivo.canRead())
```

```
                System.out.println("ok para leitura (canRead)  
");
```

```
            else
```

```
                System.out.println("Não está ok para  
leitura");
```

```
            If (meuArquivo.canWrite())
```

```
                System.out.println("ok para escrita  
(canWrite");
```

```
            else
```

```
        System.out.println("Não está 🚫 para  
escrita");  
  
        System.out.println("caminho:  
"+meuArquivo.getAbsolutePath());  
  
        System.out.println("Nome do arquivo:  
"+meuArquivo.getName());  
  
        System.out.println("Tamanho do arquivo:  
"+meuArquivo.length()+" bytes");  
  
    }else  
  
        System.out.println("Arquivo não existe");  
  
    }  
  
}
```

Resultado:

Se o arquivo não estiver disponível na pasta raiz do projeto.

A screenshot of a terminal window with a dark background. The text "Arquivo não existe" is displayed in a light-colored, monospaced font. The text is preceded by a vertical bar character "|".

Quando o arquivo estiver presente:

```
Data.txt existe  
O arquivo tem 88 bytes  
ok para leitura  
Nome do arquivo: Data.txt  
Tamanho do arquivo: 88 bytes
```



Tratamento de exceções


Várias exceções no pacote `java.io` podem ocorrer quando você estiver trabalhando com arquivos e fluxos.

- Uma exceção `FileNotFoundException` ocorre quando você tenta criar um objeto de fluxo ou arquivo usando um arquivo que não pôde ser localizado.
- Uma `EOFException` indica que o final de um arquivo foi atingido inesperadamente, pois os dados estavam sendo lidos a partir do arquivo através de um fluxo de entrada.

Essas exceções são subclasses de `IOException`. Uma maneira de lidar com todos eles é colocar todas as instruções de entrada e saída em um bloco `try-catch` que captura objetos `IOException`. Chame os métodos `toString()` ou `getMessage()` da exceção no bloco `catch` para descobrir mais sobre o problema


Lendo arquivos

Java tem um conceito de trabalhar com fluxos de dados. Você pode dizer que um programa Java lê sequências de bytes de um fluxo de

entrada (ou grava em um fluxo de saída): byte após byte, caractere após caractere, primitivo após primitivo. Por conseguinte, Java  fine vários tipos de classes que suportam fluxos, por exemplo, InputStream ou OutputStream. Existem classes especificamente destinadas à leitura de fluxos de caracteres, como Reader e Writer.

Antes de um aplicativo poder usar um arquivo de dados, ele deve abrir o arquivo. Um aplicativo Java abre um arquivo criando um objeto e associando um fluxo de bytes a esse objeto. Da mesma forma, quando você terminar de usar um arquivo, o programa deve fechar o arquivo, ou seja, torná-lo não mais disponível para o seu aplicativo.


Abaixo está uma lista de classes da biblioteca java muito importantes relacionadas ao Streams.

Classe	Descrição 
InputStream	Classe abstrata contendo métodos para executar entrada
OutputStream	Classe abstrata contendo métodos para executar saída
FileInputStream	Filho de InputStream que fornece a capacidade de ler arquivos de disco
FileOutputStream	Filho de OutputStream que fornece a capacidade de gravar em arquivos de disco
PrintStream	Filho de FilterOutputStream, que é filho de OutputStream; O PrintStream processa a saída no dispositivo de saída padrão (ou padrão) do sistema, geralmente o monitor
BufferedInputStream	Filho de FilterInputStream, que é filho de InputStream; BufferedInputStream manipula a entrada do dispositivo de entrada padrão (ou padrão) do sistema, geralmente o teclado

Ler de um arquivo

Escreveremos um programa java para ler e imprimir dados do arquivo na tela do usuário. Vamos entender como associar um objeto File ao fluxo de entrada:

- Você pode passar o nome do arquivo para o construtor da classe FileInputStream.
- Você pode criar um objeto File passando o nome do arquivo para o construtor File. Em seguida, você pode passar o objeto File para o construtor da classe FileInputStream.

O segundo método tem alguns benefícios: se você criar um  objeto File, poderá usar os métodos da classe File, como `exists()` e `lastModified()`, para recuperar informações do arquivo.

Enquanto trabalhamos com classes de fluxo, temos que cuidar de exceções verificadas. Em nosso programa, estamos fazendo isso usando um bloco try-catch.

```
package br.com.java.aula;

import java.io.*;

public class LendoArquivoDemo{

    public static void main(String[] args){

        InputStream istream;

        OutputStream ostream;

        int c;

        final int EOF = -1;

        ostream=System.out;

        try{

            File inputFile=new File("Data.txt");

            istream=new FileInputStream(inputFile);

            try{

                while((c =istream.read())!= EOF)
```

```
ostream.write@;
```



```
}catch(IOException e){
```

```
System.out.println("Error:
```

```
"+e.getMessage());
```

```
}finally{
```

```
try{
```

```
istream.close();
```

```
ostream.close();
```

```
}catch(IOException e){
```

```
System.out.println("File didnt close");
```

```
}
```

```
}
```

```
}catch(FileNotFoundException e){
```

```
e.printStackTrace();
```

```
System.exit(1);
```

```
}
```

```
}
```

```
}
```

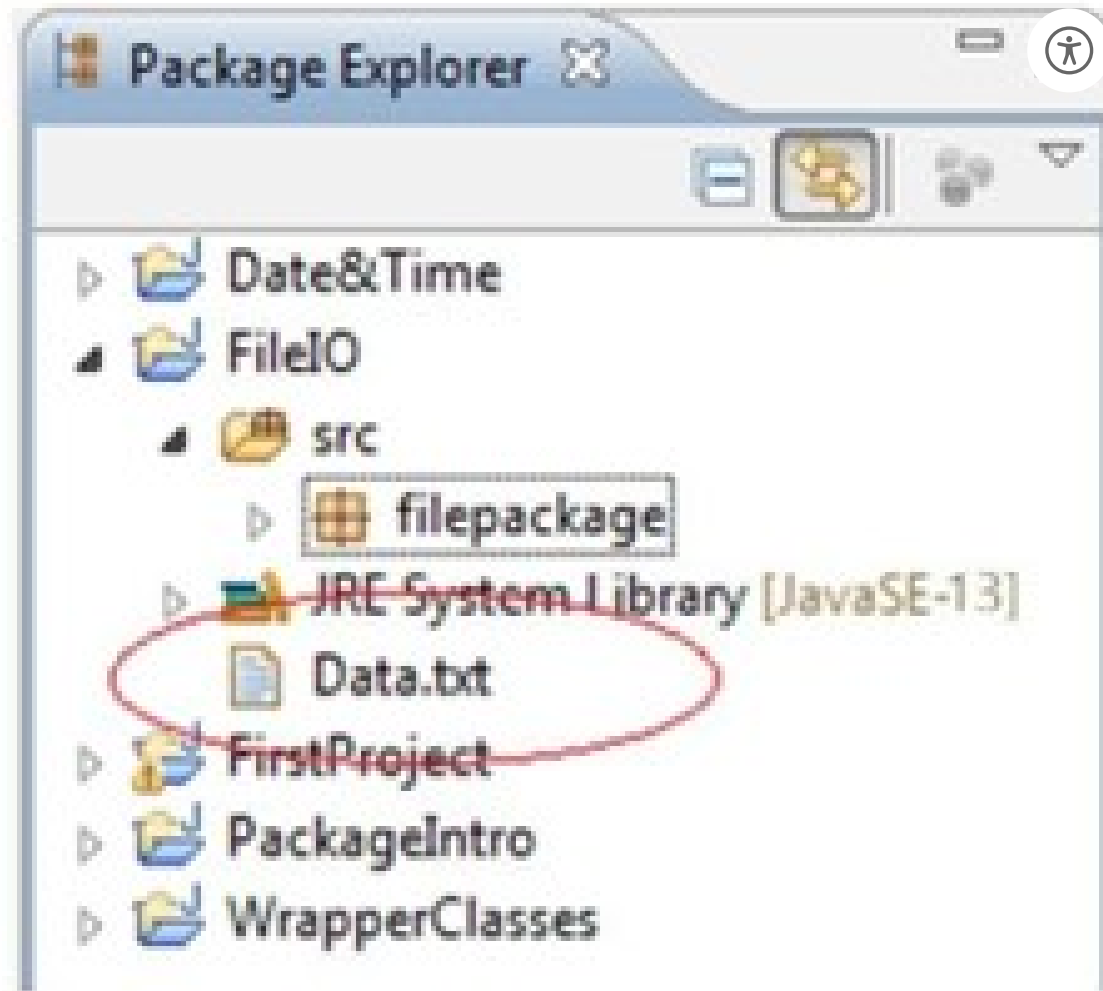
Se o arquivo estiver ausente no diretório raiz, obteremos o seguinte erro.



```
java.io.FileNotFoundException: Data.txt (The system cannot find the file specified)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:138)
  at filepackage.FileReadingDemo.main(FileReadingDemo.java:16)
```

Após criar o arquivo no diretório raiz do projeto, obteremos o conteúdo do arquivo como saída.

Localização de arquivo



Enquanto as classes de fluxo de bytes fornecem funcionalidade suficiente para manipular qualquer tipo de operação de E / S, elas não podem funcionar diretamente com caracteres Unicode. Como um dos principais objetivos do Java é oferecer suporte à filosofia “escreva uma vez, execute em qualquer lugar”, era necessário incluir suporte direto de E / S para caracteres. Agora, veremos o programa Java usando o fluxo de caracteres para ler o arquivo. Isso é muito semelhante ao `FileInputStream`, mas a JVM o trata de maneira diferente. No programa abaixo, não estamos lidando com a exceção com o bloco try-catch, mas estamos adicionando a cláusula throws

na declaração do método. A saída seria a mesma do programa acima.



```
package br.com.java.aula;

import java.io.*;

public class LendoArquivoCharacterStream{

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

        FileReader freader =new FileReader("Data.txt");

        BufferedReader br =new BufferedReader(freader);

        String s;

        while((s =br.readLine())!=null){

            System.out.println(s);

        }

        freader.close();

    }

}
```

Gravando arquivo

Podemos ler o arquivo usando fluxos. Os fluxos podem ser de dois tipos: Fluxo de caracteres ou Fluxo de bytes. Agora vamos discutir

como escrever / criar o arquivo usando o programa Java.



Gravando arquivo usando FileOutputStream

FileOutputStream cria um OutputStream que você pode usar para gravar bytes em um arquivo. Vamos entender a maneira popular de associar um objeto File ao fluxo de saída:

- Você pode passar o nome do arquivo para o construtor da classe FileOutputStream.
- Você pode criar um objeto File passando o nome do arquivo para o construtor File. Em seguida, você pode passar o objeto File para o construtor da classe FileOutputStream.

Em nosso programa, receberemos as informações do usuário usando um teclado e as escreveremos no arquivo. Após o prompt “Digite os caracteres a serem gravados no arquivo - pressione Ctrl + z para finalizar”, um bloco try mantém a instrução while. A instrução while continua sendo lida até que Ctrl + z seja pressionado pelo usuário. No caso da entrada do teclado, -1 é retornado quando você pressiona Ctrl + z.

```
pacote br.com.java.aula;
```

```
import java.io.*;
```

```
public class EscrevendoArquivoStream {
```

```
    public static void main (String [] args) {
```

```
        InputStream istream;
```



```
OutputStream ostream = null;
```

```
int c;
```

```
finalint EOF = -1;
```

```
istream = System.in;
```

```
File outFile = new file ("Data.txt");
```

```
System.out.println ("Digite caracteres para escrever no  
arquivo - pressione Ctrl + z para finalizar");
```

```
try {
```

```
    ostream = new FileOutputStream (outFile);
```

```
    while ((c = istream.read ()) != EOF)
```

```
        ostream.write c;
```

```
    } catch (IOException e) {
```

```
        System.out.println ("Erro:" + e.getMessage ());
```

```
    } finally {
```

```
        try {
```

```
            istream.close ();
```

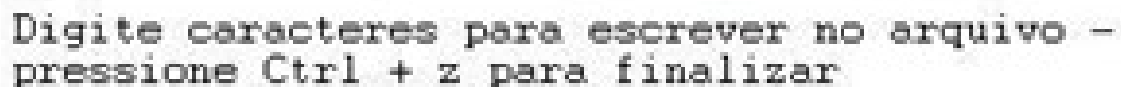
```
            ostream.close ();
```

```
        } catch (IOException e) {
```



```
        System.out.println ("O arquivo não foi  
        fechado");  
  
    }  
  
}  
  
}
```

Resultado:



```
Digite caracteres para escrever no arquivo -  
pressione Ctrl + z para finalizar
```

Gravando arquivo usando o fluxo de caracteres (FileWriter)

Enquanto as classes de fluxo de bytes fornecem funcionalidade suficiente para manipular qualquer tipo de operação de E / S, elas não podem funcionar diretamente com caracteres Unicode. Como um dos principais objetivos do Java é oferecer suporte à filosofia “escreva uma vez, execute em qualquer lugar”, era necessário incluir suporte direto de E / S para caracteres. A criação de um `FileWriter` não depende do arquivo já existente. O `FileWriter` criará o arquivo antes de abri-lo para saída quando você criar o objeto. No caso em que você tenta abrir um arquivo somente leitura, uma `IOException` será lançada.

```
package br.com.java.aula;
```



```
import java.io.*;
```

```
public class EscrevendoArquivoCharacterStream{
```

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

```
        InputStream istream;
```

```
        int c;
```

```
        finalint EOF =-1;
```

```
        istream= System.in;
```

```
        FileWriterout File=new FileWriter("Data.txt");
```

```
        BufferedWriterb Writer=new BufferedWriter(outFile);
```

```
            System.out.println("Como escrever caracteres no  
arquivo - pressione Ctrl + z para concluir");
```

```
            while((c =istream.read())!= EOF)
```

```
                bWriter.write(c);
```

```
            bWriter.close();
```

```
        }
```

```
    }
```

Processamento de arquivos Java Properties

Properties é uma extensão de arquivo usada principalmente em tecnologias relacionadas a Java para armazenar os parâmetros configuráveis de um aplicativo. Os arquivos Java Properties são recursos incríveis para adicionar informações em Java. Geralmente, esses arquivos são usados para armazenar informações estáticas no par de chaves e valor. As coisas que você não deseja codificar no código Java são inseridas nos arquivos de propriedades. A vantagem de usar o arquivo de propriedades é que podemos configurar coisas que podem sofrer alterações durante um período de tempo sem a necessidade de alterar nada no código. O arquivo de propriedades fornece flexibilidade em termos de configuração. O arquivo de propriedades de amostra é mostrado abaixo, que possui informações no par de valores-chave.

Cada parâmetro é armazenado como um par de cadeias, o lado esquerdo do sinal de igual (=) serve para armazenar o nome do parâmetro (chamado chave) e o outro para armazenar o valor.

Configuração:


Este arquivo de propriedade possui minha configuração

FileName=Data.txt

Author_Name=David

Website=java.com

Topic=Processamento do arquivo de propriedades

A primeira linha que começa com # é chamada de linha de comentário. Podemos adicionar comentários em propriedades  que serão ignoradas pelo compilador java.

Abaixo está o programa Java para ler o arquivo de propriedades acima.

```
package br.com.java.aula;

import java.io.FileInputStream;

import java.io.IOException;

import java.util.Properties;

public class LendoArquivoPropriedades{

    public static void main(String[] args){

        Propriedades prop=new Properties();

        try{

            // carregar um arquivo de propriedades para leitura

            prop.load(new FileInputStream("minhaConfig.properties"));

            // obtenha as propriedades e imprima

            prop.list(System.out);

            //Lendo cada valor de propriedade
```

```
        System.out.println(prop.getProperty("Nome do  
arquivo"));  
  
        System.out.println(prop.getProperty("Nome do  
autor"));  
  
        System.out.println(prop.getProperty("Website"));  
  
        System.out.println(prop.getProperty("TOPICO"));  
  
    }catch(IOExceptionex){  
  
        ex.printStackTrace();  
  
    }  
  
}
```

O programa usa o método `load ()` para recuperar a lista. Quando o programa é executado, ele primeiro tenta carregar a lista de um arquivo chamado `minhaConfig.properties`. Se esse arquivo existir, a lista será carregada, caso contrário a exceção de E / S será lançada.

Se o arquivo não estiver presente no local raiz do projeto padrão, obteremos uma exceção como abaixo. Também podemos fornecer um caminho completo para o arquivo.

```
java.io.FileNotFoundException: myConfig.properties (The system cannot find the file specified)  
    at java.io.FileInputStream.open(Native Method)  
    at java.io.FileInputStream.<init>(FileInputStream.java:138)  
    at java.io.FileInputStream.<init>(FileInputStream.java:97)  
    at propertiesfile.PropertyFileReading.main(PropertyFileReading.java:13)
```

Uso do método `getProperties()`



Um recurso útil da classe `Properties` é que você pode especificar uma propriedade padrão que será retornada se nenhum valor estiver associado a uma determinada chave. Por exemplo, um valor padrão pode ser especificado junto com a chave no método `getProperty()` - como `getProperty("nome", "valor padrão")`. Se o valor "nome" não for encontrado, será retornado o "valor padrão". Quando você constrói um objeto `Propriedades`, pode passar outra instância de `Propriedades` a ser usada como propriedades padrão para a nova instância.

Arquivo de propriedades de gravação

A qualquer momento, você pode gravar um objeto `Properties` em um fluxo ou lê-lo novamente. Isso torna as listas de propriedades especialmente convenientes para implementar bancos de dados simples. Por exemplo, abaixo, o programa grava estados nas capitais. Arquivo "capitals.properties" com nome do estado como chaves e capital do estado como valores.

```
package br.com.java.aula;

import java.io.FileOutputStream;

import java.io.IOException;

import java.util.Properties;

public class ArquivoEscrevendoPropriedades{

    public static void main(String args[]){
```



```
Propriedades prop=new Properties();

try{

    // set thepropertiesvalue

    prop.setProperty("Gustavo","Guilherme");

    prop.setProperty("Monica","Marcelo");

    prop.setProperty("Madhya_Pradesh","Ingrid");

    prop.setProperty("Ricardo","Jorge");

    prop.setProperty("Pablo","Melissa");

    prop.setProperty("Agatha","Lyvia");

    // salvar propriedades na pasta raiz do projeto

                                prop.store(new
FileOutputStream("capitals.properties"),null);

    }catch(IOExceptionex){

        ex.printStackTrace();

    }

}

}
```

Após a execução do programa, podemos ver um novo arquivo criado chamado "capitals.properties" na pasta raiz do projeto, como

mostrado abaixo.



Web 08 de Junho 18:03:18 IST 2020

Agatha = Lyvia

Madhyar_Pradesh = Ingrid

Monica = Marcelo

Gustavo = Guilherme

Panjab = Melissa

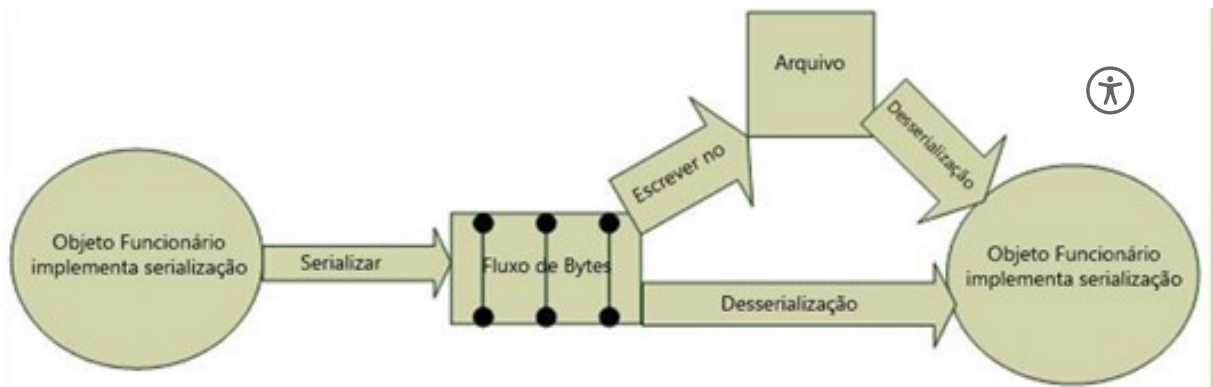
Rajastão = Jorge

Serialização Java

Criar objeto é um recurso fundamental do Java e de todas as outras linguagens orientadas a objetos. Java permite gravar objetos inteiros em arquivos, em vez de gravar separadamente campos individuais.

Imagine que você deseja salvar o estado de um ou mais objetos. Se o Java não tivesse serialização (como a versão mais antiga), seria necessário usar uma das classes de E / S para gravar o estado das variáveis de instância de todos os objetos que você deseja salvar. A serialização permite simplesmente dizer “salve este objeto e todas as suas variáveis de instância”.

Serialização é o mecanismo interno do Java para manipular objetos como fluxos de bytes; a interface serializável confere à sua classe a capacidade de ser serializada.



Trabalhando com ObjectOutputStream e ObjectInputStream

A mágica da serialização básica acontece com apenas dois métodos: um para serializar objetos e gravá-los em um fluxo e um segundo para ler o fluxo e desserializar objetos.

`ObjectOutputStream.writeObject ()` // serialize e escreva


`ObjectInputStream.readObject ()` // lê e desserializa

As classes `java.io.ObjectOutputStream` e `java.io.ObjectInputStream` são consideradas classes de nível superior no pacote `java.io` e, como aprendemos anteriormente, isso significa que você as envolverá em classes de nível inferior, como `java.io.FileOutputStream` e `java.io.FileInputStream`.

Aqui está um pequeno programa que cria um objeto (Funcionário), serializa-o e desserializa-o:

```
package br.com.java.aula;
```

```
import java.io.*;
```



```
public class SerieFuncionariosDemo{

    public static void main(String[] args){

        Funcionario c =new Funcionario("Sylvia","E123");// 2

        File outFile=new File("empSerial.ser");

        try{

                                FileOutputStream fs=new
FileOutputStream(outFile);

                                ObjectOutputStream os =new
ObjectOutputStream(fs);

                                os.writeObject();// 3

                                os.close();

        }catch(Exception e){

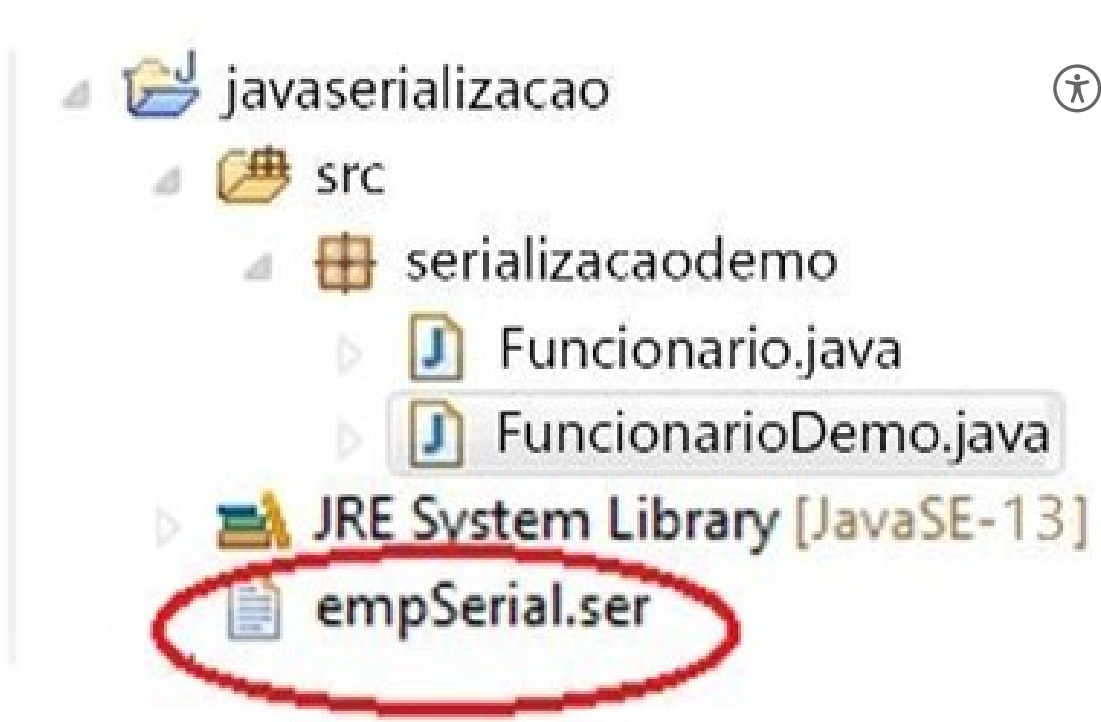
                                e.printStackTrace();

        }

    }

}
```

O programa abaixo irá ler o arquivo empSerial.ser criado pelo programa acima, que contém o estado de um objeto. O arquivo será criado no diretório raiz do projeto, como mostrado abaixo (caminho padrão, pois não mencionamos o caminho completo).



Classe Java de demonstração de desserialização

```
package br.com.java.aula;
```

```
import java.io.*;
```

```
public class DesserializaFuncionarioDemo{
```

```
    public static void main(String[] args){
```

```
        File ReadFile=new File("empSerial.ser");
```

```
        try{
```

```
            FileInputStream fis=new FileInputStream(ReadFile);
```

```
            ObjectInputStream ois=new  
            ObjectInputStream(fis);
```

```
            Funcionario e =(Funcionario)ois.readObject();
```


```
System.out.println("Nome de funcionário  
desserializado= "+e.getName());  
  
System.out.println(" ID de funcionário  
desserializado= "+e.getId());  
  
ois.close();  
  
}catch(Exception e){  
  
    e.printStackTrace();  
  
}  
  
}  
  
}
```

Resultado:

```
Nome de funcionário desserializado= Sylvia  
ID de funcionário desserializado= E123
```

Vamos dar uma olhada nos pontos principais deste exemplo:

1. Declaramos que a classe Funcionario implementa a interface Serializable. Serializable é uma interface de marcador; não possui métodos para implementar.
2. Criamos um novo objeto Funcionario, que, como sabemos, é serializável.

3. Serializamos o objeto Funcionario c invocando o método `writeObject ()`. A chamada `ofwriteO`  `c ()` executa duas tarefas: serializar o objeto e, em seguida, gravar o objeto serializado em um arquivo.

4. Desserializamos o objeto Funcionario, invocando o método `readObject ()`. O método `ThereadObject ()` retorna um `Object`, portanto, temos que converter o objeto desserializado de volta em um objeto Funcionario.

Relacionamento de serialização com composição (Has-A)

Composição (HAS-A) significa simplesmente o uso de variáveis de instância que são referências a outros objetos. Vamos entender se a classe Funcionario possui referência de objeto de departamento e se a classe de departamento não implementa uma interface serializável, temos que declarar a referência de departamento como transitória ou obteremos uma exceção ao chamar o método `serialize`.

```
package br.com.java.aula;
```

```
import java.io.*;
```

```
public class FuncionarioComDepartamentoSerialDemo{
```

```
    public static void main(String[] args){
```

```
        // Criando objeto da classe New Funcionario
```

```
        New Funcionario c =new  
        NewFuncionario("Sylvia","E123",new Departamento());
```



```
File outFile=new File("NovoFuncSerial.ser");

try{

    FileOutputStream fs=new
FileOutputStream(outFile);

    ObjectOutputStream os =new
ObjectOutputStream(fs);

    os.writeObject@; // 3

    os.close();

}catch(Exception e){

    e.printStackTrace();

}

}

}

package br.com.java.aula;

import java.io.Serializable;

class NovoFuncionarioimplementaSerial{

    private static final long serialVersionUID=1L;

    private StringeName;

    private Stringeld;
```

```
private DepartamentodName;
```



```
public String geteName(){
```

```
    return eName;
```

```
}
```

```
public String geteld(){
```

```
    return eld;
```

```
}
```

```
public Departamento getdName(){
```

```
    return dName;
```

```
}
```

```
New Funcionario(String a,String b,Departamento d){
```

```
    this.eName= a;
```

```
    this.eld= b;
```


```
    this.dName= d;
```

```
}
```

```
}
```

Resultado:

```
java.io.NotSerializableException: serializationdemo.Department
    at java.io.ObjectOutputStream.writeObject0(Unknown Source)
    at java.io.ObjectOutputStream.defaultWriteFields(Unknown Source)
    at java.io.ObjectOutputStream.writeSerialData(Unknown Source)
    at java.io.ObjectOutputStream.writeOrdinaryObject(Unknown Source)
    at java.io.ObjectOutputStream.writeObject0(Unknown Source)
    at java.io.ObjectOutputStream.writeObject(Unknown Source)
    at serializationdemo.EmployeeWithDepartmentSerialDemo.main(EmployeeWithDepartmentSerialDemo.java:15)
```



Serialização com relação de herança

Se uma superclasse for Serializable, então, de acordo com as regras normais da interface Java, todas as subclasses dessa classe implementam automaticamente Serializable implicitamente. Em outras palavras, uma subclasse de uma classe marcada como Serializable passa no teste IS-A para Serializable e, portanto, pode ser salva sem a necessidade de marcar explicitamente a subclasse como Serializable.

Se você serializar uma coleção ou uma matriz, todos os elementos deverão ser serializáveis. Um único elemento não serializável fará com que a serialização falhe se não for declarado transitório.

A serialização não é para variáveis estáticas

Finalmente, você pode perceber que falamos SOMENTE sobre variáveis de instância, não variáveis estáticas. Isso ocorre porque variáveis estáticas não fazem parte do estado da instância, portanto não fazem parte do processo de serialização.

Atividade extra

- Vídeo: “Manipulação de Arquivo”

- Link: <https://www.youtube.com/watch?v=qFfBVPnpw4w>



Referências Bibliográficas

BARNES, D. J. KOLLING, M. **Programação orientada a objetos com java: uma introdução prática usando o bluej**. 4.ed. Pearson: 2009.

FELIX, R. (Org.). **Programação orientada a objetos**. Pearson: 2017.

MEDEIROS, L. F. de. **Banco de dados: princípios e prática**. Intersaberes: 2013;

ORACLE. Java Documentation, 2021. **Documentação oficial da plataforma Java**. Disponível em: < <https://docs.oracle.com/en/java/> >.

Ir para exercício