PERCURSO DE APRENDIZAGEM 4

Universidade de Fortaleza | EAD

PROGRAMAÇÃO ORIENTADA A OBJETOS

APLICAÇÃO DE ESTRUTURAS AVANÇADAS DE ORIENTAÇÃO A OBJETOS

Universidade de Fortaleza | EAD

APLICAÇÃO DE ESTRUTURAS AVANÇADAS DE ORIENTAÇÃO A OBJETOS

Sumário

Sumário clicável

Nesse percurso de aprendizagem vamos aprender a criar, lançar e controlar exceções pré-definidas do Java e do usuário. Ademais, iremos aprender a hierarquia de classes que definem as coleções genéricas do Java, lançando mão das estruturas lineares: *Set* e *List* e das estruturas de *Hash* e *Map*. Para tanto, vamos mostrar a interface *Collections* do Java e a interface Map. Em seguida, mostraremos como trabalhar com arquivos binários e texto em Java e, por fim, mas não menos importante, vamos mostrar como o Java trabalha com fluxo e serialização de objetos em Java. Nesse ponto evidenciaremos como gravar um objeto serializado em um banco de dados ou em um arquivo binário e depois como recuperá-los e discutir sobre os possíveis problemas de gravar o objeto em uma versão da classe e recuperá-lo em outra versão.

Olá

1. Tratamento de exceções

Antes de entrarmos em detalhes sobre tratamento de exceções em Java, vamos, genericamente, falar um pouco sobre o assunto para contextualizar o problema, uma vez que o "tratamento de exceções" do Java nada mais é uma forma de se preparar para possíveis erros imprevistos durante a execução do programa, proveniente de diversas formas, tais como: erros de lógica de acesso a dispositivos, serviços, arquivos ou sistemas externos dentre outros.

Quando se cria um programa em Java ou em qualquer outra linguagem existe a possibilidade de erros ou *warnings* e as exceções entram exatamente nesse ponto. Quando ocorre algo imprevisto, as *exceptions* podem ser aliadas respeitáveis.

É importante notar que elas também podem ser utilizadas em situações previstas, por exemplo, uma operação bancária não ter saldo é uma possibilidade muito plausível e as exceções podem ajudar nesse contexto. Ou seja, não se vincula exceções a erros "imprevistos" ou "raros" elas, também, podem ser usadas para situações de uso comuns, normalmente, mapeadas antigamente pelos fluxos de exceções dos casos de usos e, agora, pelas regras de negócios das "estória de uso".

1.1 Entendendo as exceções

Uma das situações que podemos lançar mãos das exceções são provenientes de erros ou acesso a recursos que talvez não estejam disponíveis, como:

- Tentar abrir um arquivo em uma pasta que a aplicação não tenha permissão de leitura ou escrita;
- Tentar fazer consulta a um banco de dados que não está disponível;
- Tentar acessar um serviço sem credenciais corretas ou que tenha expirado;
- Não ter acesso a um banco de dados. Isso ocorre bastante em aplicações reais, em ambientes de produção. Não ter acesso ao IP do banco de dados. Normalmente, os servidores de produção precisam de uma lista de IP que permite acesso ao servidor.
- Temos outros possíveis erros de lógica que podem gerar exceções:
- Tentar manipular um objeto que está com o valor nulo;
- Dividir um número por zero, ver figura 2;
- Tentar manipular um tipo de dado.

Exemplificando um erro real ao servidor de autenticação de um serviço em ambiente produção:

Exemplo ilustrativo de um erro de acesso a um serviço:

```
22:32:33,480 ERROR [stderr] (http-thread-pool-threads - 13) java.
io.IOException: Server returned HTTP response code: 403 for URL: https://
apigateway.conectagov.estaleiro.servidor.producao/oauth2/jwt-token

22:32:33,481 ERROR [stderr] (http-thread-pool-threads - 13)       at sun.
net.www.protocol.http.HttpURLConnection.getInputStream0(HttpURLConnection.
java:1876)

22:32:33,482 ERROR [stderr] (http-thread-pool-threads - 13)at sun.net.
www.protocol.https.HttpsURLConnectionImplgetInputStream(HttpsURLConnectionImpl.
java:254)
```

No erro de acesso descrito acima, podemos verificar o código de resposta 403 que significa acesso negado. A aplicação poderia "capturar" esse tipo de erro e mostrar

uma mensagem para usuário da aplicação de “acesso negado, por favor, verifique suas credenciais de acesso ao serviço em questão”.

Outro tipo de exceção seria uma divisão zero, conforme o exemplo da figura 2. Isso ocorre quando tentamos fazer uma divisão na qual o denominador é zero.

Exemplo ilustrativo da exceção: *ArithmeticException*

```
Exception in thread “main” java.lang.ArithmeticException: / by zero
at Main.main(Main.java:4)
exit status 1
```

Para ilustrar outro tipo de exceção, temos o exemplo acima da exceção de referente a acessar uma posição de memória não existente no vetor: ArrayIndexOutOfBounds Exception. O código a seguir, sendo executado, gera a exceção mostrada no exemplo ilustrativo da exceção ArrayIndexOutOfBoundsException:

```
class Main {

 public static void main(String[] args) {
   int v[] = {1,2,3};
   System.out.println(v[3]);
 }

}
```

Exemplo ilustrativo da exceção ArrayIndexOutOfBoundsException:

```
Exception in thread “main” java.lang.ArrayIndexOutOfBoundsException: Index 3
out of bounds for length 3
at Main.main(Main.java:5)
exit status 1
```

No Exemplo ilustrativo da exceção NullPointerException, temos, com certeza, a exceção com maior ocorrência entre todas as outras, NullPointerException, a qual ocorre toda vez que tentamos manipula um objeto e ele está apontando para uma referência nula.

Exemplo ilustrativo da exceção NullPointerException:

```
Exception in thread “main” java.lang.NullPointerException
at Main.main(Main.java:7)
exit status 1
```

1.2 Tratando exceções

Uma maneira de tentar contornar os erros ou imprevistos da execução de um determinado código, conforme mostrado na seção 1.1, é tratando as exceções. No caso do Java, capturando-os e, em seguida, fazendo um tratamento específico ou de forma mais genérica.

Para tanto, na linguagem Java, usam-se os comandos *try* e *catch*.

```
try{
        //trecho de código que pode vir a lançar uma exceção
} catch(tipo_exceçao_1 e) {
        //ação a ser tomada
} catch(tipo_exceçao_2 e) {
        //ação a ser tomada
}catch(tipo_exceçao_n e)

      //ação a ser tomada
}finally {
      //ação a ser tomada

}
```

Em que, no bloco: try{ … } - Neste bloco, são introduzidas todas as linhas de código que podem vir a lançar uma exceção e catch(tipo_excessao e) { … } - Neste bloco, é descrita a ação que ocorrerá quando a exceção for capturada e o finally o código sempre será executado independente se uma exceção foi gerada ou não durante a execução do sistema. Para facilitar o entendimento, vamos codar :-).

Por exemplo, executando o código demonstrado abaixo, teríamos a seguinte saída no console:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4
out of bounds for length 3
at Main.main (Main.java:6)
exit status 1
```

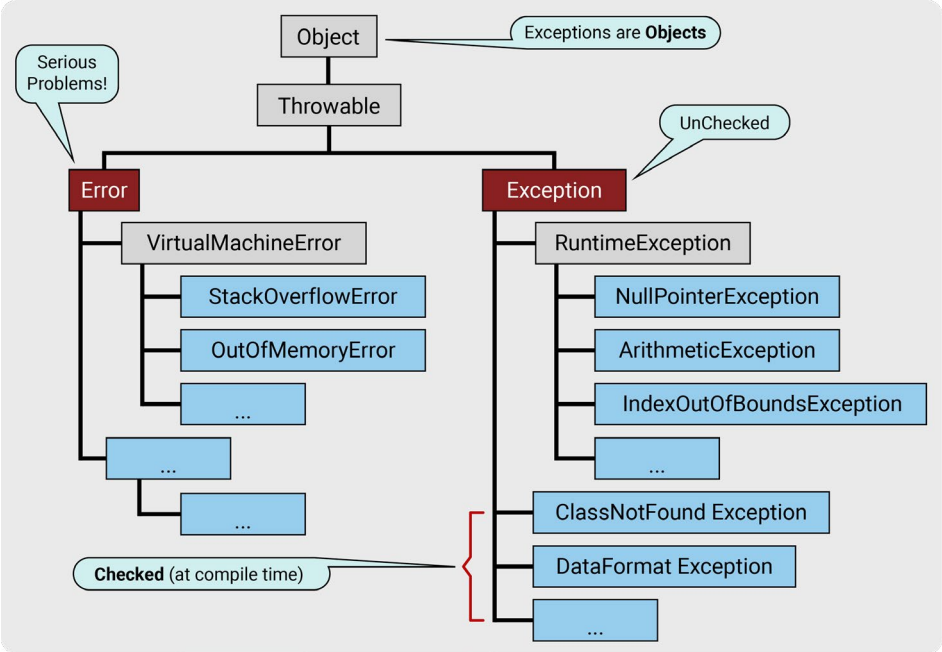
Exemplo de código para ilustrar o tratamento de exceções:

```
class Main {
 public static void main(String[] args) {
   int v[] = {1,2,3};
   try {
     System.out.println(v[4]);
   } catch(ArithmeticException e) {
     System.out.println("Acesso a posição de memória inválida!");
   }
 }
}
```

Uma vez que a captura da exceção está errada, teríamos de capturar a exceção ArrayIndexOutOfBoundsException ou uma exceção mais genérica (superclasse)

dessa exceção. Por isso, é muito importante conhecer a hierarquia de exceções mostrada no exemplo citado acima. Dessa forma, poderíamos substituir ArrayIndexOutOfBoundsException por RunTimeException, Exception ou Throwable.

Figura 1 - Hierarquia de exceções do Java



Fonte: Adaptado de UNIVERSIDADE JAVA (2021), por EaD/Unifor.

1.3 Comandos throw e throws

São situações nas quais, no método implementado, não é possível tratar a exceção lançada (gerada), isso só ocorrerá no método de origem, ou seja, aquele que está chamando o método em questão. Desse modo, temos o comando *throws* para indicar que aquele método pode gerar exceções ou exceções (basta colocar eles entre vírgulas).

Agora, imagine que desejamos lançar uma exceção específica ou até mesmo uma implementada por nós, veja a seção 1.4. Nesse contexto, usamos o comando *throw* para lançar uma exceção.

Veja o exemplo:

```
public class TesteExcecao {

  public static void main(String args[]) throws SemLetraBException  {
            String frase = “Sou um teste!”;
            if(!frase.contains(“b”) || !frase.contains(“B”))
                  throw new SemLetraBException();

  }

}
```

No qual definimos que o método **main** pode gerar uma exceção *SemLetraB Exception*, a qual é gerada caso a *String* frase não contenha a letra “B”.

1.4 Criando e utilizando exceções

Assim como qualquer objeto em Java, também é possível criar suas próprias exceções, basta criar uma classe derivada de *Exception* ou *RunTimeException*, na qual, se a classe for derivada de Exception, a exceção será *checked* e, caso contrário, *unchecked*. Qual a diferença entre elas? A primeira vai ser “checada” em tempo de compilação, enquanto a segunda, apenas em tempo de execução.

Caso seja necessário gerar uma exceção para que o usuário informe uma senha errada, como o exemplo a seguir:

```
import java.util.*;
import java.io.IOException;
public class Main {
   public static class SenhaInvalidaException extends Exception {
       public SenhaInvalidaException()  {
               super(“Senha Invalida”);
       }
   }
   public static class Autenticacao {
       int senhasistema;
       public Autenticacao(int senhasistema) {
          this.senhasistema = senhasistema;
       }
       public boolean checkSenha(int senha) {
           if(senha == this.senhasistema){
               return true;
           } else{
               return false;
           }
       }
   }
}
```

Nesse exemplo, vemos a criação de uma exceção: *SenhaInvalidaException* derivada de Exception e podemos gerar uma exceção conforme o exemplo abaixo:

```
if (autenticacao.checkSenha(n)) {
                System.out.println("Acesso Permitido");
} else throw new SenhaInvalidaException();
```

na qual "lançamos" uma exceção `throw new SenhaInvalidaException();` através do comando throw.

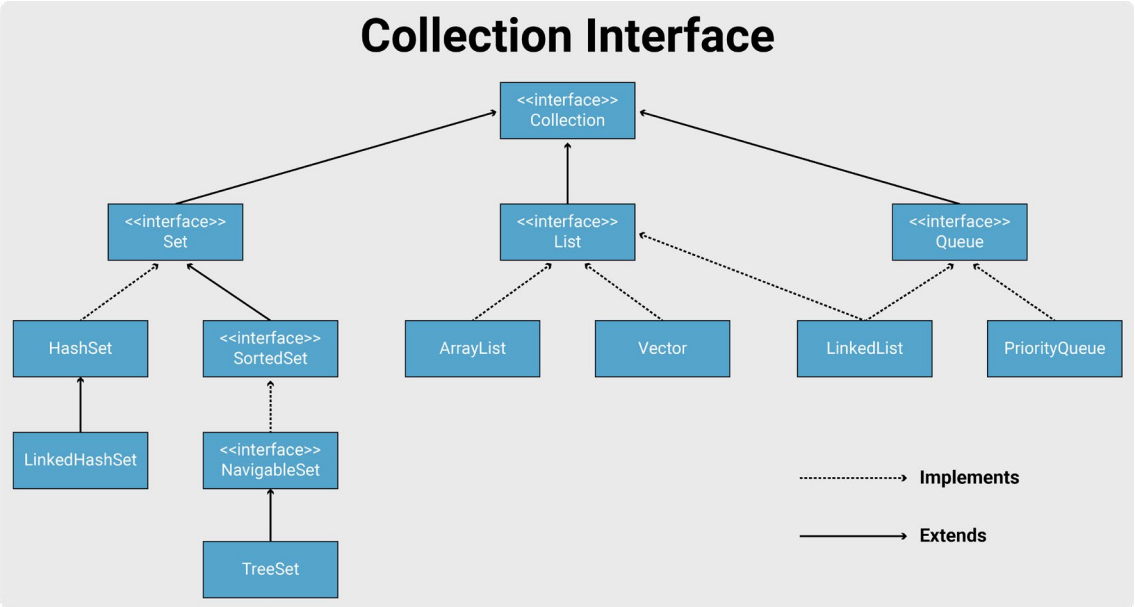
Nesse circuito de estudo, vimos que as exceções são simples de serem implementadas em Java e o usuário tem a possibilidade de criar a sua própria hierarquia de exceções e tratá-las de forma mais organizada e adaptável.

2. Coleções genéricas

Como visto na disciplina de raciocínio lógico e algoritmos, uma fase do desenvolvimento é armazenar as informações em uma estrutura de dados.

No primeiro momento, trabalhamos diretamente com *arrays*, mas eles possuem diversas limitações, sendo uma delas a de um array não ser redimensionado. Além disso, adicionar ou remover elementos deles requer deslocamento dos dados e processamento. Diante disso, a biblioteca do Java implementa diversas estruturas de dados que facilitam o trabalho do desenvolvedor. A *Figura 2* mostra a árvore da hierarquia de interfaces e classes da Java Collections que são derivadas da interface Collection.

Figura 2 - Collections em Java



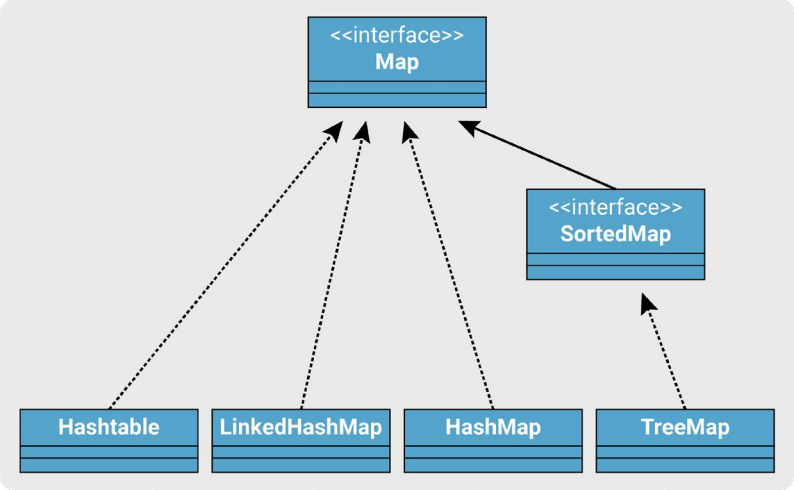
Fonte: Adaptado de DEVMEDIA (2021), por EaD/Unifor.

No diagrama UML, podemos verificar que existem três interfaces derivadas diretamente de Collection: Set (conjunto), List (Lista) e Queue (Fila). *List e Queue* representam estruturas lineares as quais a primeira armazena um conjunto de dados que é possível inserir ou deletar elementos de qualquer posição sem restrição de acesso. Já a Queue (Fila) segue o critério FIFO (First In First out) - primeiro a entrar é o primeiro a sair, com uma fila normal do dia a dia. Tem ainda uma implementação de fila de prioridade, na qual, além do tempo, são levados em conta outros critérios para ordem dos elementos que entram e saem da fila.

A interface Set define uma coleção que não permite elementos duplicados, já a *SortedSet* possibilita a classificação natural dos elementos, tal como a ordem alfabética. Por sua vez, List define uma coleção ordenada, podendo conter elementos duplicados. Em geral, prefira essa interface quando precisar de acesso aleatório, através do índice do elemento.

Além das *Collections,* temos também a hierarquia de mapas que são estruturas de dados baseadas em chave valor, como mostra a *Figura 3*. Essas interfaces, mesmo não sendo consideradas coleções, podem ser manipuladas como tal.

Figura 3 - Hierarquia de mapas



Fonte: Adaptado de DEVMEDIA (2021), por EaD/Unifor.

A interface Map, como o próprio nome indica, mapeia chaves para valores, as quais são chaves únicas. *SortedMap* é um interface que estende Map e permite a classificação ordenada de chaves.

Para facilitar o entendimento de *Collections* e *Maps*, vamos dividi-los por seção.

É importante notar que um bom desenvolvedor precisa saber qual melhor estrutura de dados (*Collections*) utilizar para cada situação, a fim de ter melhor desempenho da aplicação.

2.1 Qual melhor interface utilizar

Uma pergunta óbvia seria: qual interface utilizar dentre as sete apresentadas: *Collection*, *List*, *Set*, *SortedSet*, *Map*, *SortedMap* e *Queue*?

Para tanto, temos que analisar o problema e verificar como ele se enquadra nas características de cada uma dessas estruturas de dados.

Para facilitar, exemplificaremos: suponha que queremos manter o nome dos professores de uma universidade e acessar esses nomes em ordem de inserção. Como podemos ter homônimos, dois professores chamados Paulo, por exemplo, então a interface *List*

seria a mais adequada. Agora, suponha que vamos armazenar o número de matrícula dos professores, ou seja, uma informação que não será repetida. Nesse caso a interface *Set* é mais adequada e, por fim, caso seja necessário armazenar a matrícula e o nome do professor correspondente, teríamos que lançar mão da interface *Map*.

Como podemos ver, dependendo da abordagem, o mesmo problema pode alterar a *Collecions* mais adequada para utilização. Sendo assim, vamos apresentá-las a seguir:

2.2 Interface List

Como vimos na *Figura 3,* a interface *List* é implementada por três classes: *ArrayList*, *LinkedList* e *Vector*, ou seja, os três códigos abaixo estão corretos:

```
ArrayList arrayList = new ArrayList () ;                    //(1)
LinkedList linkedList = new LinkedList () ;                 //(2)
Vector vector = new Vector () ;                             //(3)
```

ou usando polimorfismo:

```
List arrayList = new ArrayList () ;          //(1)
List linkedList = new LinkedList () ;        //(2)
List vector = new Vector () ;                //(3)
```

O seu propósito é armazenar um conjunto de elementos em ordem linear, no qual Vector e ArrayList usam vetor para armazenar a lista, enquanto LinkedList utiliza Lista dinâmica. Os principais métodos da interface *Lista* são os responsáveis pelo "CRUD" dessa estrutura: *add* (adicionar elementos), *remove* (remover elementos), *get* (retornar elementos da lista), *set* (alterar elementos da lista).

No exemplo a seguir, podemos ver a utilização dos principais comandos da interface *List*. Inicialmente, usamos o comando *add* que adiciona o elemento no final da lista, em seguida, lançamos mão do comando *remove* que elimina o elemento da posição ZERO (primeira posição), nesse caso, teríamos a possibilidade de remover, também, usando o objeto "lista.remove("João da Silva");". Utilizamos o get(lista.size()-1) para retornar o elemento da última posição da lista e, por fim, o comando *set* para alterar o valor de uma posição da lista set(0, "Francisco Brasil"), ou seja, alteramos o valor do primeiro elemento.

```
import java.util.*;

public class Main {

  public static void main(String[] args) {

    List<String> lista = new ArrayList<String>();

    lista.add("João da Silva");

    lista.add("Antonio Sousa");

    lista.add("Lúcia Ferreira");

    System.out.println(lista);

    Collections.sort(lista);
```

```
    System.out.println(lista);
    lista.remove(0);
    lista.set(0, "Francisco Brasil");
    System.out.println(lista.get(lista.size()-1));
  }
}
```

Para armazenar elementos não repetidos e/ou ordenados, temos a interface Set.

2.3 Interface Set

Uma das características de *List* é que ela permite elementos duplicados, o que em muitos casos não é desejado, como por exemplo armazenar uma lista de empregados. Para esses casos, temos a interface *Set*, pois não permite elementos duplicados. Existem duas classes que implementam essa *HashSet* e *TreeSet*. Para facilitar o entendimento dessa Collections, vamos para o exemplo a seguir:

```
import java.util.*;
class Ex1Set {
  public static void main(String[] args) {
    Set<String> lista1 = new TreeSet<String>();
    lista1.add("Lula");
    lista1.add("Bolsonaro");
    lista1.add("Marina");
    lista1.add("Bolos");
    lista1.add("Bolos");
    lista1.add("Ciro");
    lista1.add("Daciolo");
    System.out.println(lista1);
    //operações básicas: add, remove, set, get, size
    System.out.println(lista1.contains("Daciolo"));
    }
}
```

No exemplo da classe ***Ex1Set***, temos a mesma operação *add* a diferença é que, nessa estrutura, o candidato "Bolos" só será inserido uma vez já que é um elemento repetido. A ordem de impressão diferente da interface *List* não é "ordenada" do primeiro para último e, como na *List,* temos a operação *contains* para verificar se um objeto pertence ou não ao conjunto.

A seguir, vamos entrar na interface Queue.

2.4 Interface Queue

A interface *Queue* é implementada pelas classes *LinkedList* e *PriorityQueue*, utilizando o TAD-Fila como padrão de acordo com o critério de acesso FIFO (primeiro a entrar é o primeiro a sair) como uma fila normal de banco ou supermercado. A *Tabela 1* mostra os principais métodos dessa interface.

Tabela 1 - Principais métodos da interface Queue.

Método	Descrição
add() ou offer()	Insere o elemento na Fila e retorna true se a operação for bem sucedida.
element() ou peek()	Retorno elemento do início da Fila. A diferença entre os métodos é que element lança uma exceção caso a Queue (Fila) esteja vazia e peek() retorna null.
remove() e poll()	Remove o elemento do início da Fila. O remove lança exceção caso a Fila esteja vazia e poll() retorna null.
isEmpty()	Retorna verdadeiro se a fila estiver vazia e false caso contrário.
size()	Retorna o número de elementos da Fila.

Fonte: Adaptado de INTERFACE JAVA QUEUE (2022), por EaD/Unifor.

2.5 Interface Map

Vamos supor que precisamos acessar um objeto a partir de uma valor chave, por exemplo, pelo nome do aluno acessar todos os seus dados (objeto) ou a partir de um código de barra acessar todas as informações de um boleto bancário ou ainda pelo número de um partido político acessar as informações dos candidatos, a estrutura de dados mais adequada seria Map que não estende Map, dessa forma, tem seus próprios métodos para inserir/buscar/remover elementos na estrutura.

Esta interface mapeia chaves para valores. No exemplo abaixo, a chave será o nome do aluno e o valor será o objeto aluno.

Para usar uma classe que implementa Map, quaisquer classes que forem utilizadas como chave devem sobrescrever os métodos hashCode() e equals(), conforme a classe Aluno. Isso é necessário porque em um Map as chaves não podem ser duplicadas, apesar dos valores poderem ser. Para a implementação mostrada na classe *MapaAluno*, utilizamos um TreeMap, que garante que as chaves estarão em ordem ascendente.

```
public class Aluno implements Comparable<Aluno>{

     private String nome;
     private String curso;
     double nota;

  Aluno(String nome, String curso, double nota) {
    this.nome = nome;
    this.curso = curso;
    this.nota = nota;
  }
  public String toString() {
    return this.nome;
  }

  public int compareTo(Aluno aluno) {
    return this.nome.compareTo(aluno.getNome());
  }
 public boolean equals(Object o) {
    Aluno a = (Aluno) o;
    return this.nome.equals(a.getNome());
  }
  public int hashCode() {
    return this.nome.hashCode();
  }  // Métodos getters e setters


  public String getNome() {
    return this.nome;
  }
}


import java.util.*;

public class MapaAluno {
  public static void main(String[] args) {

    Map<String, Aluno> mapa = new TreeMap<String, Aluno>();

    Aluno a = new Aluno("João da Silva", "Linux básico", 0);
    Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
    Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
    Aluno d = new Aluno("Benedito Silva", "OpenOffice", 0);


    mapa.put("João da Silva", a);
    mapa.put("Antonio Sousa", b);
```

```
        mapa.put("Lúcia Ferreira", c);
        mapa.put("Benedito Silva", d);
        System.out.println(mapa);
        System.out.println(mapa.get("Lúcia Ferreira"));

        Collection<Aluno> alunos = mapa.values();
        for (Aluno e : alunos) {
          System.out.println(e);
        }
    }
}
```

Fonte: Adaptado de DEVMEDIA, (2021)

No código acima, vimos que, na interface *Map*, informa dois tipos String e Aluno, dos quais o primeiro é a chave do mapa e o Aluno é o objeto referenciado pela chave. O método que insere na estrutura é o *put* e, para recuperar objetos, usamos *get* passando a chave como parâmetro.

3. Arquivos

Para armazenar informações de forma não volátil em um programa Java, é necessário utilizarmos banco de dados ou arquivos em um *filesystem*. Nesse circuito de aprendizagem, vamos abordar o segundo tema, sendo que a linguagem de programação nos proporciona diversos meios de se manipular arquivos. A classe *java.io.File* é responsável por representar arquivos ou pastas em um sistema de arquivos do sistema operacional. Esta classe pode oferecer informações sobre os arquivos, criá-los ou removê-los.

A linguagem Java oferece uma classe específica para a leitura do fluxo de bytes que compõem o arquivo, esta classe chama-se java.io.FileInputStream. A FileInputStream recebe em seu construtor uma referência *File*, ou uma *String* que deve representar o caminho completo do arquivo, dessa forma podemos ler as informações que estão dentro do arquivo. Antes de abordarmos a FileInputStream, vamos ver um exemplo da utilização da classe File.

Importante notar que a instanciação de um *File* não implica na criação do arquivo. Um File apenas representa um arquivo que pode existir ou não.

O construtor da classe File é o *pathname* do arquivo e existem um conjunto de métodos úteis para sua utilização:

- boolean exists() - verificar se um arquivo existe ou não no filesystem;
- boolean isDirectory - se é uma pasta válida no sistema de arquivos;
- boolean isFile - se é um arquivo válido no sistema de arquivos;
- File [] listFiles() - retorna os arquivos de um determinada pasta;
- boolean delete() - deleta um arquivo e retorna true caso a operação seja bem sucedida;
- boolean mkdir() - remove uma pasta;

A seguir, temos um exemplo da classe File, utilizando os métodos listados acima:

```
import java.io.*;

public class ApagaTeste {
            public static void main(String[] args) {
                  File file = new File("c:\\temp\\dados.dat");
                  if(file.exists() && file.isFile())    file.delete();
                  File dir = new File("c:\\temp\\dados")
                  if(!dir.exists())  file.mkdir();
            }
      }
```

No código da *class* ApagaTeste, vemos que o arquivo *dados.dat da pasta c:\temp* será apagado caso ele exista e que o diretório (pasta) c:\temp\dados será criada.

Temos, também, a classe RandomAccessFile que permite a leitura e escrita em arquivos com acesso randômico. Para tanto temos os métodos evidenciados na tabela 2.

Tabela 2 - Principais métodos da classe RandomAccessFile.

Método	Descrição
RandomAccessFile(String file,String mode)	Construtor da classe para criar um RandomFile a partir de um caminho absoluto. O modo pode ser **r** para leitura e **rw** para leitura e escrita.
RandomAccessFile(File file,String mode)	Construtor da classe para criar um RandomFile a partir de um File. O modo pode ser **r** para leitura e **rw** para leitura e escrita.
long getFilePointer()	Retorna a posição atual do ponteiro no arquivo.
long length()	Retorna o tamanho do arquivo.
void seek(long p)	Caminha **p** posição no arquivo.
int readInt(), double readDouble() ...	Ler qualquer tipo primitivo do arquivo de acesso randômico.

writeInt(int i), writeDouble(-double d) ...	Escreve qualquer tipo primitivo em um arquivo de acesso randômico.
close()	Fechar o arquivo.

Fonte: Adaptado de DEVMEDIA, (2021)

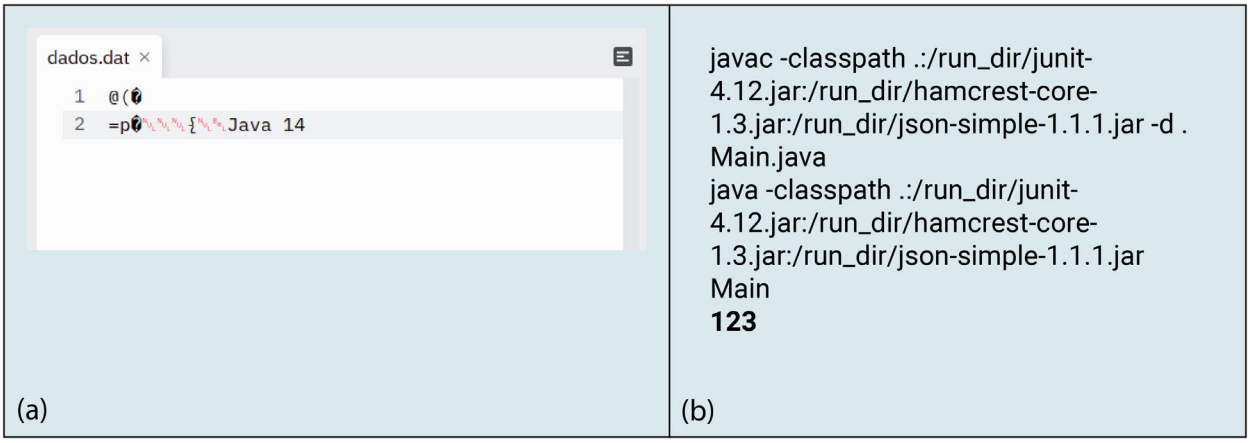
O arquivo é criado no momento da construção da classe. Se o nome não for válido, será lançado um exception: *FileNotFoundException*. Com esse objeto, é possível navegar pelos bytes do arquivo. Vamos "codar" para ficar mais claro esse tipo de manipulação de arquivos randômicos.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

class AcessoArquivoRandomico {
      public static void main(String[] args) {
            RandomAccessFile r = null;
            File file = new File("dados.dat");
            try{                          r = new RandomAccessFile(file,"rw");
                  r.writeDouble(12.42);// Escreve 8 bytes
                  r.writeInt(123);    // Escreve 4 bytes
                  r.seek(8);           // Vai até a posição do 9 bytes
              int i = r.readInt();   // Lê 4 bytes e transforma em int
              System.out.println(i);
              r.writeUTF("Java 14");
            }catch(FileNotFoundException e){
                  System.out.println("Arquivo não pode ser criado");
            }catch(IOException e){
                  System.out.println("Erro ao acessar arquivo"+file);
            } finally{
                  r.close();
            }
      }
}
```

Na classe *AcessoArquivoRandomico*, criamos o arquivo **dados.dat** e escrevemos tipos primitivos double e int, em seguida, lemos do arquivo o tipo primitivo int, após pular os 8 bytes do tipo long e, por fim, escrevermos a string "Java 14" no arquivo binário. A seguir, mostramos na figura 4 (a) o arquivo gravado no sistema de arquivos e na (b) o que foi impresso na tela.

Figura 4 - (a) arquivo dados.bat e a impressão da execução do código da classe *AcessoArquivoRandomico*



Fonte: Elaborada pelo autor (2022), produzido por EaD/Unifor.

Agora vamos utilizar um objeto de *FileInputStream*(String pathcaminho) para ler um arquivo texto, para tanto, vamos criar um objeto a partir do caminho do arquivo e uma variável c inteira para receber caracter por caracter do arquivo até que o arquivo seja totalmente lido (o método read retorne -1). Vamos ao exemplo:

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

/**
 * Classe utilizada para ler as informações de um arquivo e imprimir
 * no console.
 */

public class ExemploFileInputStream {
  public static void main(String[] args) {
    ExemploFileInputStream exemplo = new ExemploFileInputStream();
    exemplo.lerArquivo("C:\\Arquivos\\teste.txt");
  }
  public void lerArquivo(String caminho) {
    FileInputStream fis = null;
    try {      /* Cria um objeto do tipo FileInputStream para ler o arquivo.
*/
      fis = new FileInputStream(caminho);
      /* Cria uma variavel interia para ler os caracteres do arquivo,
        lembrando que todo caractere no Java é representado por um numero
inteiro. */
        int c;
      /* Le o caracter do arquivo e guarda na variavel inteira c, quando
        o caracter lido for igual a -1, significa que chegou ao final do
```

```
arquivo. */
      while((c = fis.read()) != -1) {         System.out.println((char) c);
}
    } catch (FileNotFoundException ex) {
      System.out.println("Erro ao abrir o arquivo.");
    } catch (IOException ex) {
      System.out.println("Erro ao ler o arquivo.");
    } finally {
      try {
        if(fis != null) {
          fis.close();            /* Fecha o arquivo que está sendo lido. */
        }
      } catch (IOException ex) {
        System.out.println("Erro ao fechar o arquivo.");
      }
    }
  }
}
```

Fonte: (UNIVERSIDADE JAVA, 2022)

Vamos agora gravar arquivo binário usando as classes FileOutputStream e DataOutputStream. Os métodos writeUTF(), writeChar(), writeInt e writeDouble() serão utilizados de forma semelhante ao *RandomAccessFile*.

```
import java.io.DataOutputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Scanner;
public class Exemplo2 {
  public static void main(String[] args) throws FileNotFoundException,
IOException {
    Scanner ler = new Scanner(System.in);
    String nome;    char sexo;    int idade, altura;
    double pc; // peso corporal
    FileOutputStream arq = new FileOutputStream("d:\\arquivo.dat");
    DataOutputStream gravarArq = new DataOutputStream(arq);
    System.out.printf("Informe o seu nome:\n");
    nome = ler.nextLine();
    System.out.printf("\nInforme o seu sexo (M/F)..........: ");
    sexo = (char)System.in.read();
    System.out.printf("Informe a sua idade...............: ");
    idade = ler.nextInt();
    System.out.printf("Informe o seu peso corporal (em kg): ");
    pc = ler.nextDouble();
    System.out.printf("Informe a sua altura (em cm).......: ");
    altura = ler.nextInt();
```

```
    gravarArq.writeUTF(nome);     gravarArq.writeChar(sexo);
    gravarArq.writeInt(idade);     gravarArq.writeDouble(pc);
    gravarArq.writeInt(altura);
 arq.close();   }
}
```

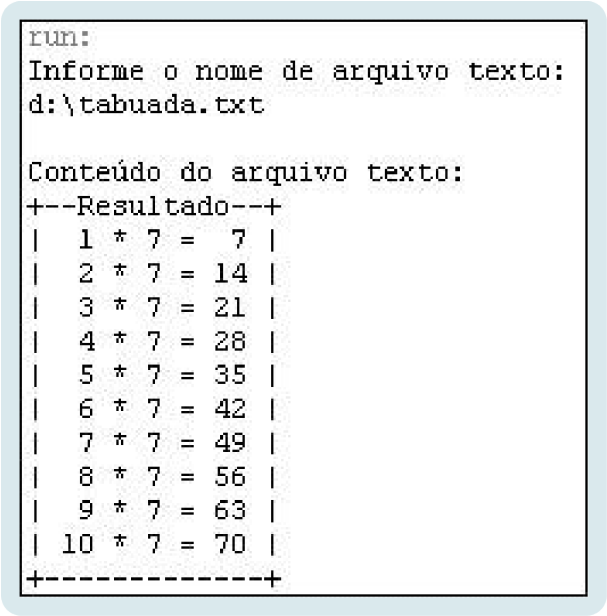
Nesse ponto, vemos uma relação entre esse circuito de aprendizagem com o próximo referente ao fluxo de dados (*stream*) e serialização de objetos. Para gravar as informações em um arquivo binário, precisamos utilizar as classes de *Streams* do Java. Para tanto, utilizamos a classe java.io.InputStream a qual recebe os dados de forma primitiva em bytes. Por esse motivo, existem diversas implementações em Java de subclasses dessa classe como a classe java.io.FileInputStream. Já a classe OutputStream é uma classe para ler bytes de forma primitiva de um arquivo e por esse motivo existem diversas classes derivadas dela, como por exemplo: java.io.FileOutputStream.

Para escrever um arquivo texto sequencialmente, vamos utilizar as classes FileWriter e PrinterWriter na criação e gravação de dados baseados em caracteres de um arquivo texto. Nesse contexto, vamos utilizar os métodos print(), println() e printf() são aplicados na gravação caracteres. No exemplo abaixo, iremos escrever a tabuada em um arquivo texto.

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;
public class GravarArquivoTexto {
  public static void main(String[] args) throws IOException {
    Scanner ler = new Scanner(System.in);      int i, n;
    System.out.printf("Informe o número para a tabuada:\n");  n = ler.
nextInt();
    FileWriter arq = new FileWriter("d:\\tabuada.txt");
    PrintWriter gravarArq = new PrintWriter(arq);
    gravarArq.printf("+--Resultado--+%n");
    for (i=1; i<=10; i++) {  gravarArq.printf("| %2d * %d = %2d |%n", i, n,
(i*n)); }
    gravarArq.printf("+--------------+%n");
   arq.close(); System.out.printf("\nTabuada do %d foi gravada  \"d:\\tabuada.
txt\".\n",n);
  }
}
```

Se o usuário informar a tabuada de 7, por exemplo, o arquivo gravado será semelhante ao da *Figura 5*.

Figura 5 - Resultado do arquivo texto gerado pelo código da classe



Fonte: Adaptado de DEVMEDIA (2021), por EaD/Unifor.

No próximo circuito de aprendizagem, vamos entender melhor a utilização de Stream e serialização de objetos e, por conseguinte, poderemos entender melhor como o Java trabalha com arquivos. Para o entendimento mais detalhado desse circuito de aprendizagem, é importante a leitura do próximo #vamosemfrente.

4. Fluxos e serialização de objetos

Um *Stream* nada mais é do que uma sequência de dados/caracteres. Como visto no circuito de aprendizagem 3, os Input Stream são usados para leitura de dados e os Output Stream são usados para escrever dados que podem ser armazenados na rede ou no banco de dados. Como visto anteriormente, as classes derivadas dessas duas classes podem ser utilizadas para gravar ou ler arquivos.

Como exemplo, vamos mostrar a utilização de FileWriter e *FileReader* para fazer cópias entre arquivos. Vamos ao exemplo:

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Reader;
```

```
import java.io.Writer;
import java.util.Scanner;
/** * Classe utilizada para ler um arquivo e copia-lo. */
public class ExemploWriter {
  public static void main(String[] args) {
    ExemploWriter exemplo = new ExemploWriter();
    Scanner s = new Scanner(System.in);
    System.out.print("Digite o caminho do arquivo de entrada: ");
    String entrada = s.nextLine();
    System.out.print("Digite o caminho do arquivo de saida: ");
    String saida = s.nextLine();
    exemplo.copiarArquivo(entrada, saida);
  }
  public void copiarArquivo(String entrada, String saida) {
    Reader reader = null;
    Writer writer = null;
    try {
      reader = new FileReader(entrada);
      writer = new FileWriter(saida);
      int c;
      while((c = reader.read()) != -1) {
        System.out.print((char) c);
        writer.write(c);
      }
      System.out.println("\n");
    } catch (FileNotFoundException ex) {
      System.out.println(entrada + " ou " + saida + " nao existem!");
    } catch (IOException ex) {
      System.out.println("Erro de leitura de arquivo!");
    } finally {
      try {
        if(reader != null) {
          reader.close();
        }
      } catch (IOException ex) {
        System.out.println("Erro ao fechar o arquivo " + entrada);
      }
      try {
        if(writer != null)  writer.close();
      } catch (IOException ex) {
        System.out.println("Erro ao fechar o arquivo " + saida);
      }
    }
  }
}
```

Fonte: (UNIVERSIDADE JAVA, 2022)

Outro exemplo de Stream de alto nível são os PrintWriter, os quais podemos utilizar para escrever arquivos textos. Para facilitar o entendimento, temos Stream de baixo nível que implementam a escrita e leitura de bytes nos dispositivos apropriados e Stream de alto nível que tratam de tipos de mais alto nível, como tipos primitivos em Java, e usam stream de baixo nível para escrever ou ler os bytes necessários. Abaixo temos o exemplo de escrita em arquivo texto:

```
import java.io.File;
      import java.io.FileNotFoundException;
      import java.io.FileWriter;
      import java.io.IOException;
      import java.io.PrintWriter;
      class Main {
            public static void main(String[] args) {
                  File file = new File("texto.txt");
                  FileWriter fiw = null;
                  PrintWriter pw = null;
                  try{                        fiw = new FileWriter(file);  pw =
new PrintWriter(fiw);
                      pw.println("Escrevendo arquivos com Java");
                      pw.println("É muito fácil !! 123");
                      pw.append("append no arquivo .. ");
                  } catch(FileNotFoundException e){
                        System.out.println("Arquivo não foi encontrado");
                  }catch(IOException e){
                        System.out.println("Erro ao acessar arquivo"+file);
                  } finally{
                        pw.close(); //Fechando primeiro pw
                  }
            }
      }
```

Um exemplo agora de cópia de arquivos usando Stream de baixo nível. Para tanto, vamos à prática:

```
import java.io.*;
public class CopyFile {
   public static void main(String args[]) throws IOException   {
      FileInputStream in = null;
      FileOutputStream out = null;
      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");
         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      } finally {
         if (in != null) {
```

```
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Fonte: (NÚCLEO JAVA, 2015)

O código da classe *CopyFile* exemplifica bem a utilização de Stream de baixo nível para cópia de arquivos dessa forma temos dois exemplos, um utilizando Stream de alto nível, mais aconselhável e outra usando Stream de baixo nível para fazer a mesma *feature*.

Os bytes de uma stream de baixo nível muitas vezes constituem int, doubles, Strings e etc, enquanto Streams de alto nível não leem de dispositivos, mas sim de Streams de baixo nível. As Streams de alto nível possuem como classes bases FilterInputStream e FilterOutputStream.

A classe DataInputStream é bastante usada para leitura enquanto a classe DataOutputStream é utilizada para escrita, como visto no exemplo Exemplo2 do circuito de aprendizagem anterior. No exemplo abaixo, temos a leitura de arquivo usando DataInputStream:

```
File file = new File("c:\\dados.dat");
try{
      FileInputStream fis = new FileInputStream(file);
      DataInputStream dis = new DataInputStream(fis);
      //lendo
      double d = dis.readDouble();
      int i = dis.readInt();
      String s = dis.readUTF();
      //fechando a stream
} catch(FileNotFoundException e){
       System.out.println("Arquivo não foi encontrado");
}catch(IOException e){
       System.out.println("Erro ao acessar arquivo"+file);
} finally{
      dis.close(); //Fechando primeiro dis
      fis.close();

}
```

Para exemplificar os métodos da classe *DataInputStream* evidenciamos, na Tabela 3:

Tabela 3 - Principais métodos da classe DataInputStream.

Método	Descrição
DataInputStream (InputStream instream)	Construtor da classe para criar um RandomFile a partir de um caminho absoluto. O modo pode ser r para leitura e rw para leitura e escrita.
boolean readBoolean()	Ler uma variável do tipo *boolean*.
char readChar()	Ler uma variável do tipo *char*.
double readDouble()	Ler uma variável do tipo *double*.
float readFloat()	Ler uma variável do tipo *float*.
long readLong()	Ler uma variável do tipo *long*.
short readShort()	Ler uma variável do tipo *short*.
String readUTF()	Ler uma variável do tipo *String*.

Fonte: (UNIFOR EaD, 2022)

No Java, temos as opções de outras Streams de alto nível como BufferedInputStream e BufferedOutputStream nas quais essas duas classes criam objetos para leitura e escrita de blocos grandes de uma só vez diminuindo o funil (*overhead*) *input* e *output*.

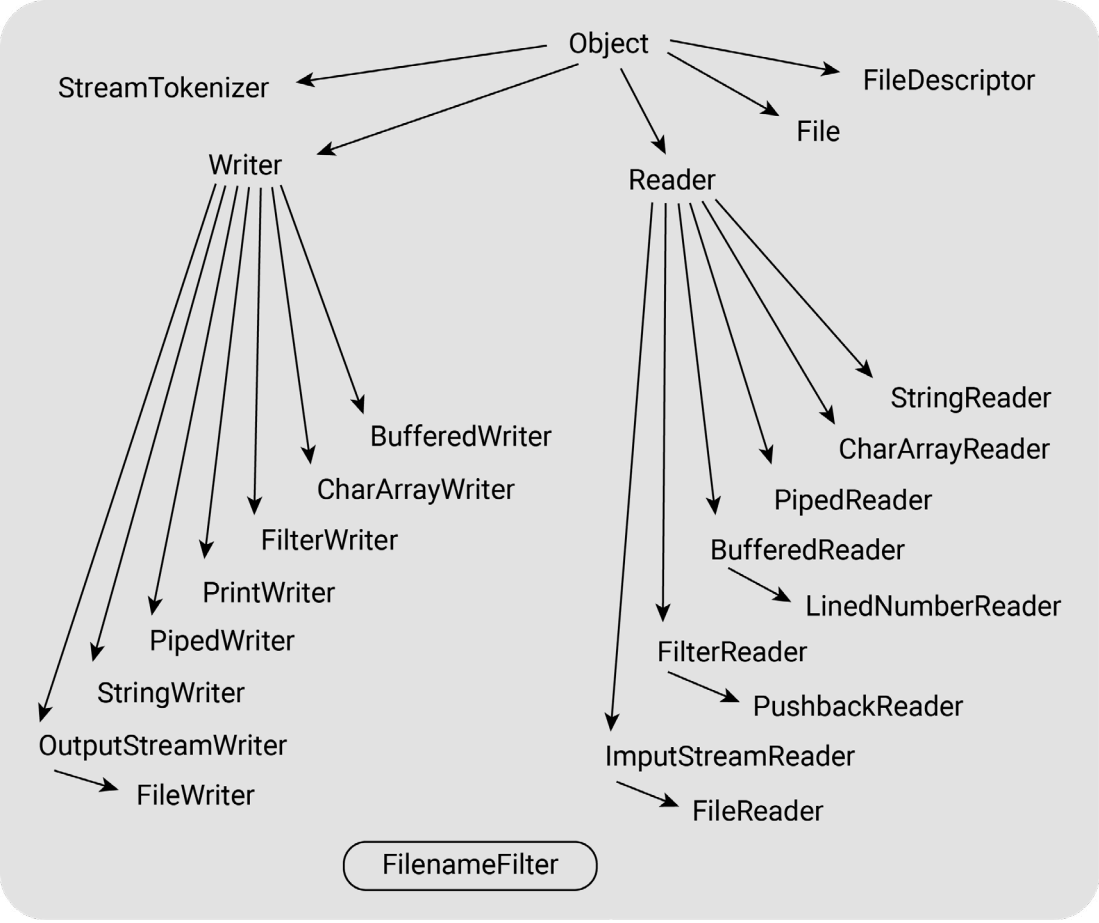
Outros exemplos são PrintStream usada para escrever tipos primitivos convertendo-os em caracteres, exemplo: System.out e System.err e o *PushbackInput Stream* que permite que os bytes recém lidos sejam colocados de volta como se nunca tivessem sido lidos.

A seguir um exemplo de código que utiliza três dessas classes de Stream de alto nível, criando uma cadeia de streams:

```
FileInputStream fis = new FileInputStream(file);
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

Temos, também, as classes Readers e Writers de streams de baixo e alto nível, que são exclusivamente orientadas a caracteres. As de baixos níveis são: *FileReader*, *FileWriter*, *PipedReader*, *PipedWriter*, *StringReader*, *StringWriter* e as de alto nível são: *BufferedReader*, *BufferedWriter*, *InputStreamReader*, *OutputStreamWriter*, *LineNumberReader* e *PrintWriter*. A *Figura 6* mostra a hierarquia das classes baseada em *Character* e na *Figura 7* baseada em *Byte*.

Figura 6 - Hierarquia das classes de Stream *Character*.



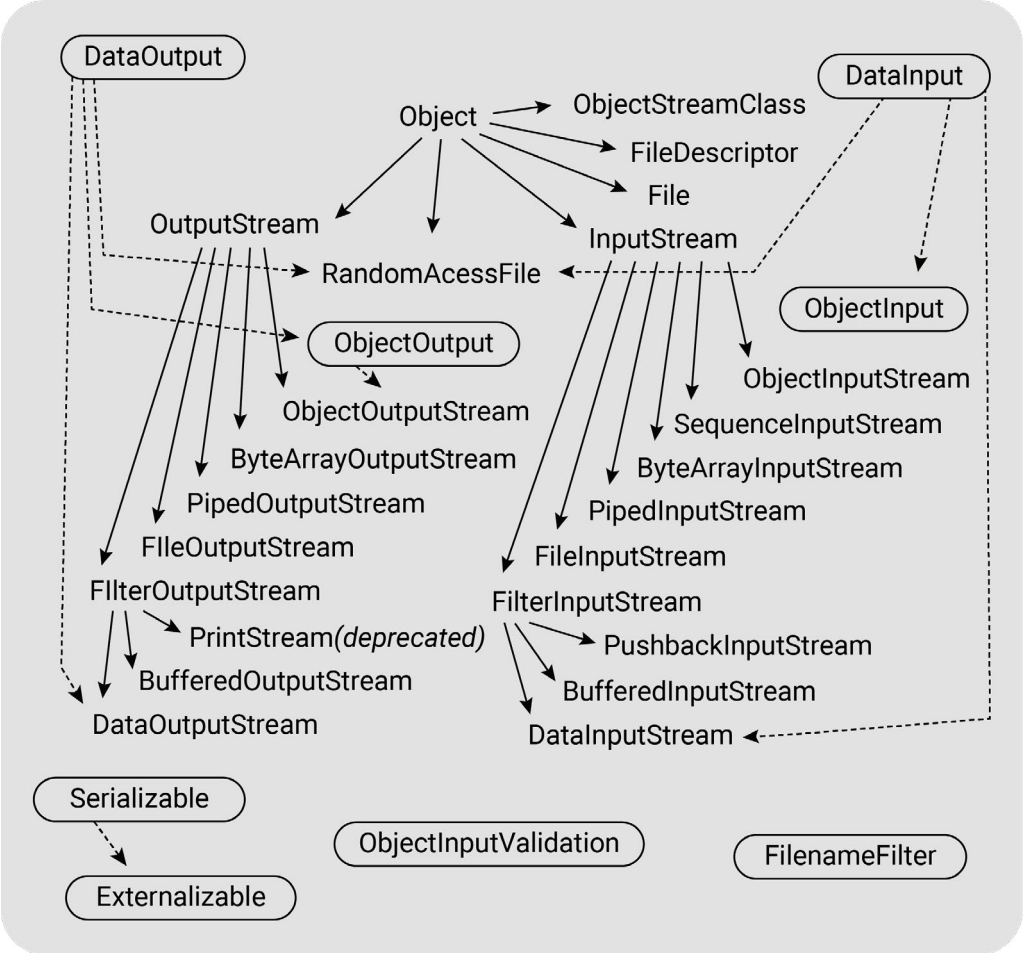
Fonte: (PACOTE JAVA.IO, 2022).

As figuras 6 e 7 mostram bem como o Java tem um leque de opções bem interessante para trabalhar com Stream de baixo e alto nível. Como vimos, temos a opção de trabalhar com arquivos textos e binários, além de trabalhar diretamente com bytes ou caracteres, além de utilizar buffer para diminuir o overhead de I/O.

Por fim, vamos tratar sobre serialização de objetos, que, além da escrita de tipos primitivos, Java também permite a escrita e leitura de objetos em streams. Para esse processo, damos o nome de serialização, ou seja, a transformação de objetos em bytes. Para serializar um objeto, deve-se usar os streams de alto nível: ObjectOutputStream e ObjectInputStream.

Nesse processo de serialização, apenas os dados do objeto são serializados e temos que ter o cuidado para quando ler o objeto a classe seja a mesma do momento da gravação, por isso, é importante “marcar” a versão das classes Java.

Figura 7 - Hierarquia das classes de *Stream Byte*.



Fonte:(PACOTE JAVA.IO, 2022).

Como exemplo, temos o código abaixo que grava o Vetor MyVector no sistema de arquivos.

```
File file = new File("c:\\obj.dat");
Vector myVector = new Vector();
myVector.add(new Color(123,12,234));
myVectory.add(new Color(111,32,131));

try{
      FileOutputStream fos = new FileOutputStream(file);
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(myVector);
      oos.close();
      fos.close();
}catch (IOException e){
      System.out.println("Erro ao escrever arquivo:"+file);
}
```

Outro ponto importante é que a serialização é recursiva e todos os objetos referenciados podem ser serializados.

Vemos que o ObjectOutputStream é derivado de FileOutputStream e, por meio do método writeObject, serializamos o objeto. Como desafio, faça um código para ler o arquivo obj. dat e recuperá-lo em um Lista em memória.

Com a serialização, podemos verificar a importância do *serialVersionUID* que identifica a versão da classe que foi usada durante o processo de serialização. Esse valor é utilizado para rastrear a compatibilidade de versões serializadas das classes e saber se o objeto que se está recuperando é de uma versão "compatível" com a versão da classe que foi utilizada na origem para serializar o objeto: ou seja, os arquivos. class gerados a partir da compilação da classe não precisam ser necessariamente os mesmos para que o processo de serialização ocorra com sucesso. O objetivo da presença desse atributo é identificar a versão da classe que foi criada durante o processo de serialização do objeto.

Resumo:

Iniciamos o estudo desse percurso aprendendo sobre tratamento exceções, um recurso muito poderoso da linguagem Java, o qual podemos criar nossa hierarquia de gerenciamento de erros ou alertas do sistema. Por meio dessa hierarquia, podemos definir quais regras de exceção serão tratadas e como elas poderão ser tratadas, exemplo, quais exceções deverão obrigar o *rollback* (retorno ao estado inicial) da operação e como deverá ser tratada cada exceção, além de definirmos quais erros serão deverão ser tratados em tempo de compilação ou execução. Aprendemos, também, nesse conteúdo a tratar e criar as *exception* do próprio programador. Em seguida, fomos mergulhar nas interfaces Collections e Map responsáveis em criar as estruturas de dados dos sistemas. Nesse percurso, estudamos as interfaces *List*, *Set*, *Queue* e *Map* e suas implementações. Aprendemos que, para uma lista de elementos com repetição, podemos utilizar *List*, para elementos não repetidos, usamos *Set* e *Queue*, para implementação de Fila. *Map* é a única interface dessas quatro que não pertence a *Collections* e tem a possibilidade de gravar chave valor, em que as chaves não podem ser repetidas.

No terceiro e quarto circuito de aprendizagem, entramos no mundo *Streams* (fluxos) e dos arquivos. Vimos que stream são fluxos de entrada e saída, ou seja, fluxo de dados que são bastante populares no momento por causa das plataformas de *streamings*. Como *stream,* podemos gravar ou ler informações de diversos dispositivos como redes, banco de dados e arquivos. Tentamos entender toda a hierarquia de classes dos *stream* de bytes dos caracteres, tentando entender as classes que trabalham em baixo e alto nível, sendo que as segundas usam as primeiras para fazer uma leitura e escrita de dados mais

performática, uma vez que ler ou escrever byte por byte não é viável para grande volume de dados.

Por fim, evidenciamos a serialização de objetos no qual convertemos um objeto em um fluxo de bytes para armazená-lo ou transmiti-lo para memória, banco de dados ou arquivo. Nesse ponto, podemos verificar a importância do *serialVersionUID* que identifica a versão da classe que foi usada durante o processo de serialização.

Referências

EDELWEISS, Nina; LIVI, **Maria Aparecida Castro. Algoritmos e programação com exemplos em Pascal e C**. Porto Alegre: Bookman, 2014. Disponível em: https://integrada.minhabiblioteca.com.br/books/9788582601907.(DIGITAL) (Cód.:612)

FOWLER, Martin. **UML essencial:** um breve guia para linguagem-padrão de modelagem de objetos. 3. ed. Porto Alegre: Bookman, 2011. Disponível em: https://integrada. minhabiblioteca.com.br/books/9788560031382. (DIGITAL) (Cód.:6903)

HORSTMANN, Cay. **Conceitos de computação com Java**. 5. ed. Porto Alegre: Bookman, 2009. Disponível em: https://integrada.minhabiblioteca.com.br/books/ 9788577804078. (DIGITAL) (Cód.:2813)

MANZANO, José Augusto N. G. **Programação de computadores com Java**. São Paulo: Érica, 2014. Disponível em: https://integrada.minhabiblioteca.com.br/books/ 9788536531137. (DIGITAL) (Cód.:30259)

SEBESTA, Robert W. **Conceitos de linguagens de programação**. 4. ed. Porto Alegre: Bookman, 2000.

SCHILDT, Herbert. **Java para iniciantes**: crie, compile e execute programas Java rapidamente. 6. ed. Porto Alegre: Bookman, 2015. Disponível em: https://integrada.minhabiblioteca.com.br/books/ 9788582603376. (DIGITAL) (Cód.:6269)

SOUZA, Marco A. Furlan; GOMES, Marcelo Marques; SOARES, Márcio Vieira; CONCILIO, Ricardo. **Algoritmos e lógica de programação: um texto introdutório para a engenharia.** 3ed. São Paulo: Cengage Learning, 2019

USINAGEM: o que é e qual a sua importância. www.ccvindustrial.com.br, 2019. Disponível em: <https://ccvindustrial.com.br/usinagem-o-que-e-e-qual-a-sua-importancia>. Acesso em: 22 dez 2020.

WAZLAWICK, Raul Sidnei. **Análise e design orientados a objetos para sistema de informação**: modelagem com UML, OCL e IFML. 3. ed. Rio de Janeiro: Elsevier, 2015. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/ 9788595153653/cfi/6/2!/4/2@0.00:0.00. (DIGITAL) (Cód.:30192)

WINDER, Russel; GRAHAM, Roberts (autor). **Desenvolvendo software em Java**. 3. ed. Rio de Janeiro: LTC, 2009. Disponível em: https://integrada.minhabiblioteca.com. br/books/978-85-216-1994-9. (DIGITAL) (Cód.:3517)

UNIVERSIDADE DE FORTALEZA (UNIFOR)

Presidência
Lenise Queiroz Rocha

Vice-Presidência
Manoela Queiroz Bacelar

Reitoria
Fátima Maria Fernandes Veras

Vice-Reitoria de Ensino de Graduação e Pós-Graduação
Maria Clara Cavalcante Bugarim

Vice-Reitoria de Pesquisa
José Milton de Sousa Filho

Vice-Reitoria de Extensão
Randal Martins Pompeu

Vice-Reitoria de Administração
José Maria Gondim Felismino Júnior

Diretoria de Comunicação e Marketing
Ana Leopoldina M. Quezado V. Vale

Diretoria de Planejamento
Marcelo Nogueira Magalhães

Diretoria de Tecnologia
José Eurico de Vasconcelos Filho

Diretoria do Centro de Ciências da Comunicação e Gestão
Danielle Batista Coimbra

Diretoria do Centro de Ciências da Saúde
Lia Maria Brasil de Souza Barroso

Diretoria do Centro de Ciências Jurídicas
Katherinne de Macêdo Maciel Mihaliuc

Diretoria do Centro de Ciências Tecnológicas
Jackson Sávio de Vasconcelos Silva

AUTOR
MAIKOL MAGALHÃES RODRIGUES

Possui graduação em Ciência da Computação pela Universidade Estadual do Ceará (1998) e mestrado em Ciência da Computação pela Universidade Estadual de Campinas (2001). Atualmente é analista de sistemas - Serviço Federal de Processamento de Dados, professor assistente da Faculdade Farias Brito e professor assistênte da Universidade de Fortaleza. Tem experiência na área de Ciência da Computação, com ênfase em Modelos Analíticos e de Simulação, atuando principalmente nos seguintes temas: programação linear inteira, programação matemática, otimização combinatória, softwares de otimização e modelos de programação linear.

RESPONSABILIDADE TÉCNICA

UNIVERSIDADE DE FORTALEZA ENSINANDO E APRENDENDO — VRE Vice-Reitoria de Ensino de Graduação e Pós-Graduação

Universidade de Fortaleza | EAD

COORDENAÇÃO DA EDUCAÇÃO A DISTÂNCIA

Coordenação Geral de EAD
Douglas Royer

Coordenação de Ensino e Recursos EAD
Andrea Chagas Alves de Almeida

Supervisão de Ensino e Aprendizagem
Carla Dolores Menezes de Oliveira

Supervisão de Planejamento Educacional
Ana Flávia Beviláqua Melo

Supervisão de Recursos EAD
Andrea Chagas Alves de Almeida

Supervisão de Operações e Atendimento
Mírian Cristina de Lima

Analista Educacional
Lara Meneses Saldanha Nepomuceno

Projeto Instrucional
Igor Gomes Rebouças

Revisão Gramatical
José Ferreira Silva Bastos

Identidade Visual / Arte
Francisco Cristiano Lopes de Sousa

Editoração / Diagramação
Régis da Silva Pereira

Produção de Áudio e Vídeo
Pedro Henrique de Moura Mendes

Programação / Implementação
Francisco Weslley Lima