
Newton
Quem se prepara, não para.

Newton
Quem se prepara, não para.

Linguagens de Programação

3º período

Prof. Dr. João Paulo Aramuni

Sumário

Newton
Quem se prepara, não para.

- Java
  - I/O em Java
  - Lendo e gravando Arquivos
  - Fluxo de Objetos

Sumário

Newton
Quem se prepara, não para.

- Java
  - **I/O em Java**
  - Lendo e gravando Arquivos
  - Fluxo de Objetos

Java

Newton
Quem se prepara, não para.

- I/O em Java é baseado em fluxos:
  - Os programas Java executam I/O por intermédio de fluxos.
  - Um fluxo de I/O é uma abstração que produz ou consome informações. Ele é vinculado a um dispositivo físico pelo sistema I/O Java.
  - Java implementa os fluxos de I/O dentro de hierarquias de classes definidas no pacote java.io.

Java

Newton
Quem se prepara, não para.

- I/O em Java é baseado em fluxos:
  - Os fluxos de I/O em Java podem ser bytes ou caracteres.
  - No nível mais baixo, todo I/O continua orientado a bytes.
  - Os fluxos baseados em caracteres apenas fornecem um meio conveniente e eficiente de tratamento de caracteres.

Classes de fluxo de byte

Classe de fluxo de bytes	Significado
BufferedInputStream	Fluxo de entrada armazenado em buffer
BufferedOutputStream	Fluxo de saída armazenado em buffer
ByteArrayInputStream	Fluxo de entrada que lê de um array de bytes
ByteArrayOutputStream	Fluxo de saída que grava em um array de bytes
DataInputStream	Fluxo de entrada que contém métodos para a leitura dos tipos de dados padrão Java
DataOutputStream	Fluxo de saída que contém métodos para a gravação dos tipos de dados padrão Java
FileInputStream	Fluxo de entrada que lê de um arquivo
FileOutputStream	Fluxo de saída que grava em um arquivo
FilterInputStream	Implementa **InputStream**
FilterOutputStream	Implementa **OutputStream**
InputStream	Classe abstrata que descreve a entrada em fluxo
ObjectInputStream	Fluxo de entrada para objetos
ObjectOutputStream	Fluxo de saída para objetos
OutputStream	Classe abstrata que descreve a saída em fluxo
PipedInputStream	Pipe de entrada
PipedOutputStream	Pip de saída
PrintStream	Fluxo de saída que contém **print( )** e **println( )**
PushbackInputStream	Fluxo de entrada que permite que bytes sejam retornados para o fluxo

Newton
Quem se prepara, não para.

No topo delas estão duas classes abstratas:

InputStream e **OutputStream**.

InputStream é usada para entrada e OutputStream para saída.

Classes de fluxo de caracteres

Classe de fluxo de caracteres	Significado
BufferedReader	Fluxo de caractere de entrada armazenado em buffer
BufferedWriter	Fluxo de caractere de saída armazenado em buffer
CharArrayReader	Fluxo de entrada que lê de um array de caracteres
CharArrayWriter	Fluxo de saída que grava em um array de caracteres
FileReader	Fluxo de entrada que lê de um arquivo
FileWriter	Fluxo de saída que grava em um arquivo
FilterReader	Leitor filtrado
FilterWriter	Gravador filtrado
InputStreamReader	Fluxo de entrada que converte bytes em caracteres
LineNumberReader	Fluxo de entrada que conta linhas
OutputStreamWriter	Fluxo de saída que converte caracteres em bytes
PipedReader	Pipe de entrada
PipedWriter	Pipe de saída
PrintWriter	Fluxo de saída que contém **print( )** e **println( )**
PushbackReader	Fluxo de entrada que permite que caracteres sejam retornados para o fluxo
Reader	Classe abstrata que descreve a entrada de caracteres em fluxo
StringReader	Fluxo de entrada que lê de um string
StringWriter	Fluxo de saída que grava em um string
Writer	Classe abstrata que descreve a saída de caracteres em fluxo

Newton
Quem se prepara, não para.

No topo delas estão duas classes abstratas:

Reader e Writer.

Reader é usada para entrada e Writer para saída.

Java

Newton
Quem se prepara, não para.

- Fluxos pré-definidos:
  - 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.

  - **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 uso o console.

Fluxos pré-definidos

Newton
Quem se prepara, não para.

Tabela 10-3 Métodos definidos por **InputStream**

Método	Descrição
int available( )	Retorna o número de bytes de entrada atualmente disponíveis para leitura.
void close( )	Fecha a origem da entrada. Tentativas de leitura adicionais gerarão uma **IOException**.
void mark(int *numBytes*)	Insere uma marca no ponto atual do fluxo de entrada que permanecerá válida até *numBytes* bytes serem lidos.
boolean markSupported( )	Retorna **true** se **mark( )**/**reset( )** tiverem suporte no fluxo chamador.
int read( )	Retorna uma representação em inteiros do próximo byte disponível da entrada. É retornado –1 quando o fim do fluxo é alcançado.
int read(byte *buffer*[ ])	Tenta ler até *buffer.length* bytes em *buffer* e retorna o número de bytes que foram lidos com sucesso. É retornado –1 quando o fim do fluxo é alcançado.
int read(byte *buffer*[ ], int *deslocamento*, int *numBytes*)	Tenta ler até *numBytes* bytes em *buffer* começando em *buffer*[deslocamento] e retornando o número de bytes lidos com sucesso. É retornado –1 quando o fim do fluxo é alcançado.
void reset( )	Volta o ponteiro da entrada à marca definida anteriormente.
long skip(long *numBytes*)	Ignora (isto é, salta) *numBytes* bytes da entrada, retornando o número de bytes ignorados.

Tabela 10-4 Métodos definidos por **OutputStream**

Método	Descrição
void close( )	Fecha o fluxo de saída. Tentativas de gravação adicionais gerarão uma **IOException**.
void flush( )	Faz qualquer saída que tiver sido armazenada em buffer ser enviada para seu destino, isto é, esvazia o buffer de saída.
void write(int *b*)	Grava um único byte em um fluxo de saída. Observe que o parâmetro é um **int**, e isso permite que você chame **write( )** com expressões sem ter que convertê-las novamente para **byte**.
void write(byte *buffer*[ ])	Grava um array de bytes completo em um fluxo de saída.
void write(*byte buffer*[ ], int *deslocamento*, int *numBytes*)	Grava um subconjunto de **numBytes** bytes a partir do array **buffer**, começando em **buffer**[**deslocamento**].

- Fluxos pré-definidos:
  - Já que **System.in** é instância de **InputStream**, temos automaticamente acesso aos métodos definidos por **InputStream**.

  - Infelizmente, **InputStream** só define um método de entrada, **read()**, que lê **bytes**. Há três versões (sobrecarga de método ou overload) de **read()**, que são mostradas abaixo:
    - `int read( ) throws IOException`
    - `int read(byte dados[ ]) throws IOException`
    - `int read(byte dados[ ], int início, int max) throws IOException`

Sumário

Newton
Quem se prepara, não para.

- Java
  - I/O em Java
  - **Lendo e gravando Arquivos**
  - Fluxo de Objetos

Exemplo de classe de leitura e escrita

Newton
Quem se prepara, não para.

- O método escritor tem como parâmetro de entrada o **path** (url/caminho) do arquivo que será **escrito**.

- private static void **escritor** (String path) throws IOException:
  - Esse método tem como principal objeto interno o **BufferedWriter** que é que a classe responsável por gerar o buffer que será utilizado para realizar a escrita no arquivo .txt.

```
private static void escritor(String path) throws IOException {
    BufferedWriter buffWrite = new BufferedWriter(new FileWriter(path));
    String linha = "";
    Scanner in = new Scanner(System.in);
    System.out.println("Escreva algo: ");
    linha = in.nextLine();
    buffWrite.append(linha + "\n");
    buffWrite.close();
    in.close();
}
```

Uma parte importante deste método, responsável por inserir no arquivo txt, é exatamente o método **append** localizado no objeto **buffWrite**.
Este método tem como função inserir uma nova sequência de caracteres ao arquivo texto:
buffWrite.append(linha+"\n");

Exemplo de classe de leitura e escrita

Newton
Quem se prepara, não para.

- O método leitor tem como parâmetro de entrada o **path** (url/caminho) do arquivo que será **lido**.
- private static void **leitor** (String path) throws IOException:
  - Esse método tem como principal objeto interno o **BufferedReader** que é que a classe responsável por gerar o buffer que será utilizado para realizar a leitura do arquivo .txt.

```
private static void leitor(String path) throws IOException {
    BufferedReader buffRead = new BufferedReader(new FileReader(path));
    String linha = "";
    while (true) {
        if (linha != null) {
            System.out.println(linha);

        } else
            break;
        linha = buffRead.readLine();
    }
    buffRead.close();
}
```

Trecho do laço while que contém o método utilizado para obter o valor de uma **linha (string).**
No exemplo utilizado, o valor da linha foi atribuída a uma variável do tipo String:
linha = buffRead.readLine();

Exemplo de classe de leitura e escrita

Newton
Quem se prepara, não para.

- A classe deste projeto contém um método estático (**main**), que tem como função rodar os métodos **escritor** e **leitor**, passando como parâmetro o **path** do arquivo a ser manipulado.

```java
package meuapp;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class TesteArquivos {
    public static void main(String[] args) {
        String path = "teste.txt";
        try {
            escritor(path);
            leitor(path);
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
```

Caminho e nome do arquivo

Exemplo de classe de leitura e escrita

Newton
Quem se prepara, não para.

- Dica:
  - É de suma importância fechar os arquivos quando o método termina de utilizar os mesmos.
  - Para isso basta utilizar o método **close**, contido na classe de leitura ou escrita de arquivos.

  - **buffWrite.close(); ou buffRead.close();**

- O que pode acontecer caso o método não seja fechado?

  - Vazamento de recursos
  - Perda de dados
  - Inconsistência de dados
  - Corrupção de arquivos

Exemplo de classe de leitura e escrita

Newton
Quem se prepara, não para.

- Podemos utilizar outra classe, a **File**, para verificar a existência do arquivo .txt durante sua leitura e, caso não exista, lançarmos uma exceção FileNotFoundException.

```
private static void leitor(String path) throws IOException {
    File arquivo = new File(path);
    if (arquivo.exists()) {

        BufferedReader buffRead = new BufferedReader(new FileReader(path));
        String linha = "";
        while (true) {
            if (linha != null) {
                System.out.println(linha);

            } else
                break;
            linha = buffRead.readLine();
        }
        buffRead.close();
    } else {
        throw new FileNotFoundException("Arquivo não encontrado.");
    }
}
```

Exemplo de classe de leitura e escrita

Newton
Quem se prepara, não para.

- Outros métodos úteis da classe File:

  - canRead()
  - canWrite()
  - createNewFile()
  - delete()
  - exists()
  - getName()
  - getParentFile()
  - getPath()
  - isDirectory()
  - isFile()
  - isHidden()
  - lastModified()
  - length()
  - list()
  - listFiles()
  - mkdir()
  - mkdirs()
  - renameTo()
  - setLastModified()
  - setReadOnly()

Sumário

Newton
Quem se prepara, não para.

- Java
  - I/O em Java
  - Lendo e gravando Arquivos
  - **Fluxo de Objetos**

- Fluxo de Objetos
  - Os dados, no final das contas, são armazenados como uma série de bytes, mas pode-se pensar neles, em um nível mais alto, como sendo uma sequência de caracteres ou objetos.

  - Java fornece uma hierarquia de classes de entrada e saída cuja base são as classes **InputStream** e **OutputStream**.

  - Pode-se manipular fluxos de diversos formatos: compactados, textos, registros de tamanho fixo com acesso aleatório e **objetos**.

- Fluxo de Objetos
  - A seguir, será apresentada uma forma para se manipular fluxos de objetos usando o mecanismo de **serialização de objetos**.
    - Usando-se as classes **ObjectOutputStream** e **ObjectInputStream** pode-se gravar e ler objetos em um fluxo, de forma automática, através de um mecanismo chamado serialização de objetos.
  - Classe java.io.**ObjectOutputStream**:
    - **ObjectOutputStream** (OutputStream saída)
      - Cria um ObjectOutputStream de modo que se possa escrever objetos no OutputStream (fluxo de saída) especificado.
    - void **writeObject** (**Object** obj)
      - Escreve o objeto especificado no ObjectOutputStream.
      - A classe do objeto, a assinatura da classe e os valores dos campos não marcados como estáticos da classe são escritos e de todas as suas superclasses.

- Fluxo de Objetos
  - A seguir, será apresentada uma forma para se manipular fluxos de objetos usando o mecanismo de **serialização de objetos**.
    - Usando-se as classes **ObjectOutputStream** e **ObjectInputStream** pode-se gravar e ler objetos em um fluxo, de forma automática, através de um mecanismo chamado serialização de objetos.

  - Classe java.io.**ObjectInputStream**:
    - **ObjectInputStream** (**InputStream** entrada)
      - Cria um ObjectInputStream para ler informações de objetos do InputStream (fluxo de entrada) especificado.
    - Object **readObject** ()
      - Lê um objeto do ObjectInputStream. Lê a classe do objeto, a assinatura da classe e os valores dos campos não estáticos da classe e de todas as suas superclasses.
      - Ele faz a desserialização para permitir que múltiplas referências de objetos possam ser recuperadas.

- Fluxo de Objetos
  - Classe java.io.**FileInputStream**:
    - **FileInputStream** (**String** nome)
      - Cria um novo fluxo de entrada de arquivo, usando o arquivo cujo nome de caminho é especificado pela string nome.
    - **FileOutputStream** (**String** nome)
      - Cria um novo fluxo de saída de arquivo especificado pela string nome. Os nomes de caminhos que não são absolutos são interpretados como relativos ao diretório de trabalho.
      - **Atenção**: apaga automaticamente qualquer arquivo existente com esse nome.
    - **FileOutputStream** (**String** nome, **boolean** anexar)
      - Cria um novo fluxo de saída de arquivo especificado pela string nome.
      - Os nomes de caminhos que não são absolutos são interpretados como relativos ao diretório de trabalho. Se o parâmetro anexar for true, então os dados serão adicionados ao final do arquivo. Um arquivo existente com o mesmo nome não será apagado.

- Fluxo de Objetos
  - Passos para salvar dados de um objeto:
    - Abra um objeto da classe ObjectOutputStream
      - **ObjectOutputStream out = new ObjectOutputStream (new FileOutputStream(“empregado.dat”);**

  - Salve os objetos usando o método **writeObject**:

```
Empregado e1 = new Empregado(“João”,2500, new Day(1996,12,10));
Gerente g1 = new Gerente(“Maria”,3800, new Day(1995,10,15);
out.writeObject(e1);
out.writeObject(g1);
```

- Fluxo de Objetos
  - Passos para ler dados de um objeto:
    - Abra um objeto da classe ObjectInputStream
      - **ObjectInputStream in = new ObjectInputStream (new FileInputStream(“empregado.dat”));**

  - Leia os objetos usando o método readObject:

    Empregado e1 = (Empregado) in.readObject();
    Gerente g1 = (Gerente) in.readObject();

    - Os objetos são gravados como Object e podem ser convertidos para o seu tipo original usando cast.
    - Para cada chamada de readObject() , lê um objeto na mesma ordem em que foram salvos.
    - Para que uma classe possa ser gravada e restaurada num fluxo de objetos é necessário que ela implemente a interface **Serializable**.
      - Esta interface não tem métodos, portanto não é necessário alterar nada na classe.
      - public class Empregado **implements Serializable { . . . }**

- Fluxo de Objetos
  - Pode-se **salvar/restaurar um array de objetos** em uma única operação:

```
Empregado[ ] emp = new Empregado[3];
. . .
out.writeObject(emp);
. . .
Empregado[ ]  empNovo = (Empregado[ ]) in.readObject( );
```

  - Ao salvar um objeto que contém outros objetos, os mesmos são gravados automaticamente.
    - Quando um objeto é compartilhado por vários outros objetos, somente uma cópia é gravada.
    - A técnica de **serialização** atribui um número de série a cada objeto. Após um objeto ser gravado, uma outra cópia compartilhada apenas recebe uma referência ao número de série já gravado anteriormente.

Java

Newton
Quem se prepara, não para.

- Fluxo de Objetos
  - Serialização de objetos:

```
1  package meuapp;
2
3  import java.io.Serializable;
4
5  public class TesteObjetos implements Serializable {
6
7  }
8
```

The serializable class TesteObjetos does not declare a static final serialVersionUID field of type long

4 quick fixes available:

- Add default serial version ID
- Add generated serial version ID
- Add @SuppressWarnings 'serial' to 'TesteObjetos'
- Configure problem severity

```
3  import java.io.Serializable;
4
5  public class TesteObjetos implements Serializable {
6
7      private static final long serialVersionUID = -7980768887826716532L;
8
9  }
```

Java

Newton
Quem se prepara, não para.

- Fluxo de Objetos
  - Para dados de objetos, evite utilizar arquivos **.txt**.
    - Ao invés disso, utilize a gravação em arquivos **.dat**.

  - Vejamos agora um exemplo completo de leitura e escrita de **Objetos**.

```java
package meuapp;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Date;

public class TesteObjetos {

    public static void main(String[] args) {
        try {
            escritor();
            leitor();
        } catch (ClassNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
```

```java
private static void escritor() throws IOException {
    ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("empregado.dat"));

    Empregado e1 = new Empregado("João", 2500, new Date());
    Gerente g1 = new Gerente("Maria", 3800, new Date());
    out.writeObject(e1);
    out.writeObject(g1);

    out.close();

}

private static void leitor() throws IOException, ClassNotFoundException {

    File arquivo = new File("empregado.dat");
    if (arquivo.exists()) {

        ObjectInputStream in = new ObjectInputStream(new FileInputStream("empregado.dat"));

        Empregado e1 = (Empregado) in.readObject();
        Gerente g1 = (Gerente) in.readObject();

        System.out.println(e1.toString());
        System.out.println(g1.toString());

        in.close();
    } else {
        throw new FileNotFoundException("Arquivo não encontrado.");
    }

}
```

Newton
Quem se prepara, não para.

```java
public class Empregado implements Serializable {

    private static final long serialVersionUID = -7718847802421381774L;
    private String nome;
    private double salario;
    private Date dataEntrada;

    public Empregado(String nome, double salario, Date dataEntrada) {
        super();
        this.nome = nome;
        this.salario = salario;
        this.dataEntrada = dataEntrada;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public double getSalario() {
        return salario;
    }

    public void setSalario(double salario) {
        this.salario = salario;
    }

    public Date getDataEntrada() {
        return dataEntrada;
    }

    public void setDataEntrada(Date dataEntrada) {
        this.dataEntrada = dataEntrada;
    }

    @Override
    public String toString() {
        return "Empregado [nome=" + nome + ", salario=" + salario + ", dataEntrada=" + dataEntrada + "]";
    }
```

```java
package meuapp;

import java.io.Serializable;

public class Gerente extends Empregado implements Serializable {

    private static final long serialVersionUID = -7633327157673107837L;

    public Gerente() {
        super();
    }

    public Gerente(String nome, double salario, Date dataEntrada) {
        super(nome, salario, dataEntrada);
    }

    @Override
    public String toString() {
        return "Gerente [nome=" + getNome() + ", salario=" + getSalario() + ", dataEntrada=" + getDataEntrada() + "]";
    }
}
```

Bônus

Newton
Quem se prepara, não para.

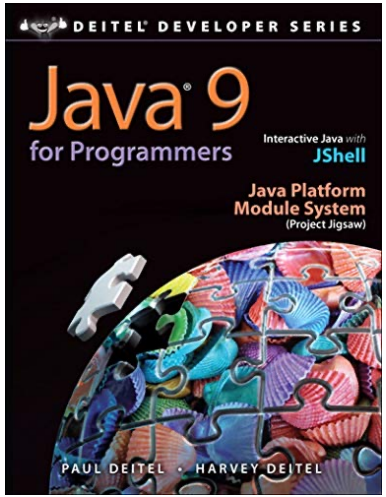
- Parâmetro vs Argumento:
  - Você sabe a diferença de um parâmetro para um argumento?

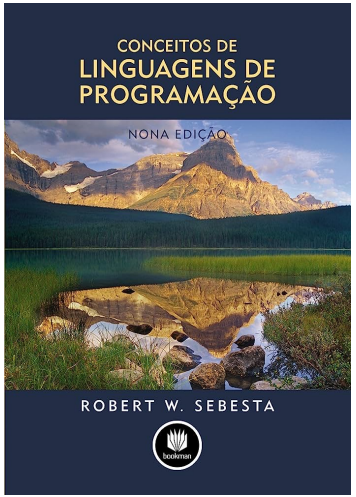
- Parâmetro:
  - Parâmetro é uma variável ou valor que é definido na declaração de um método (ou função) e que atua como um espaço reservado para um valor que deve ser passado quando o método é chamado.

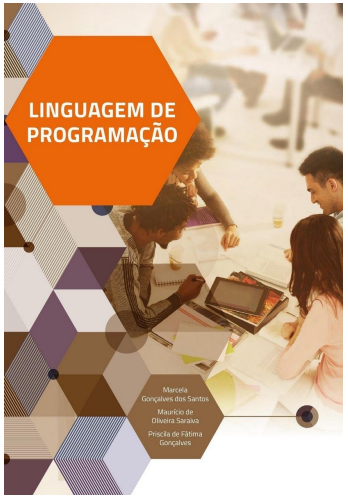
  - ```
    public void somar(int num1, int num2){
      int resultado = num1 + num2;
      System.out.println("A soma é: " + resultado);
     }
    ```
    ```

Bônus

Newton
Quem se prepara, não para.

- Parâmetro vs Argumento:
  - Você sabe a diferença de um parâmetro para um argumento?

- Argumento:
  - Argumento é o valor real que é passado para um método quando ele é invocado. Ele preenche o espaço reservado pelo parâmetro na chamada do método.

  - ```
    public static void main(String[] args) {
      int a = 5;
      int b = 3;
      somar(a, b);
      // Neste exemplo, 'a' e 'b' são os argumentos que
      // correspondem aos parâmetros 'num1' e 'num2'.
    }
    ```
    ```

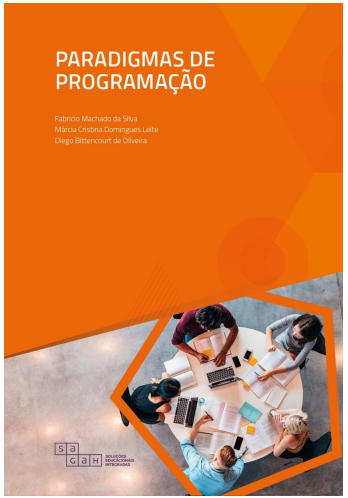
Referências

Newton
Quem se prepara, não para.









Obrigado!

joao.aramuni@newtonpaiva.br