Características das Linguagens de Programação I

Conteúdo: Tratamento de exceções e arquivos

Prof. Dsc. Giomar Sequeiros
giomar@eng.uerj.br

Exceções

Tipos de erros em programação

Existem basicamente três tipos de erros a que um programa de computador pode estar sujeito:

- Erros de Sintaxe (escrita);
- Erros de Semântica (lógica);
- Erros durante a execução (exceções).

Erros de sintaxe

- A Sintaxe diz respeito à forma como as instruções **devem ser escritas**, ou seja, o conjunto de **regras** formais que especificam a composição dos algoritmos a partir de letras, dígitos e outros símbolos.

- É o tipo mais primitivo de erro e o mais **facilmente detectado**, pois são facilmente encontrados pelos programas tradutores (interpretadores e compiladores).

Erros de sintaxe: exemplo

```
1   package media;
2
3   import java.util.Scanner;
4
5   public class Media {
6
7       public static void main(String[] args) {
8
9           Scanner entrada = new Scanner(System.in);
10
            float a = entradas.nextFloat();
            float b = entradas.nextFloat();
            float c = entradas.nextFloat();
14
            float media = A+B+C/3;
16
17          System.out.println( "A média é : " + media);
18      }
19  }
```

Erros de sintaxe: exemplo

- Corrigindo o erro:

```
1  package media;
2
3  import java.util.Scanner;
4
5  public class Media {
6
7      public static void main(String[] args) {
8
9          Scanner entrada = new Scanner(System.in);
10
11         float a = entrada.nextFloat();
12         float b = entrada.nextFloat();
13         float c = entrada.nextFloat();
14
15         float media = a+b+c /3;
16
17         System.out.println( "A média é : " + media);
18     }
19 }
```

Erros de semântica

- A Semântica diz respeito ao significado lógico das instruções que serão executadas pelo computador.
- Os erros de semântica também são conhecidos como **erros de "lógica"** do programa.
- A violação da semântica de um algoritmo **não impede** que ele seja **executado**, nem causa um erro durante sua tradução.
- Todavia, ele processará um **resultado diferente** do desejado.

Erros de semântica: exemplo

-

```
1  package media;
2
3  import java.util.Scanner;
4
5  public class Media {
6
7          public static void main(String[] args) {
8
9              Scanner entrada = new Scanner(System.in);
10
11             float a = entrada.nextFloat();
12             float b = entrada.nextFloat();
13             float c = entrada.nextFloat();
14
15             float media = a+b+c/3;
16
17             System.out.println( "A média é : " + media);
18         }
19 }
```

```
Saída - Media (run)
run:
2
4
6
A média é : 8.0
CONSTRUÍDO COM SUCESSO (tempo total: 7 segundos)
```

Erros de semântica: exemplo

- Corrigindo o erro de semântica:

```
1   package media;
2
3   import java.util.Scanner;
4
5   public class Media {
6
7       public static void main(String[] args) {
8
9           Scanner entrada = new Scanner(System.in);
10
11          float a = entrada.nextFloat();
12          float b = entrada.nextFloat();
13          float c = entrada.nextFloat();
14
15          float media = (a+b+c)/3;
16
17          System.out.println( "A média é : " + media);
18      }
19  }
```

```
Saída - Media (run)
run:

2
4
6
A média é : 4.0
CONSTRUÍDO COM SUCESSO (tempo total: 8 segundos)
```

Erros de execução (exceção)

- Os erros durante a execução dos programas são chamados de exceções.

- Em outras linguagens são chamados genericamente de “RunTime Error”, e podem ser causados por uma enormidade de circunstâncias, tais como:

  - Falta de memória;
  - Impossibilidade de gravar ou de abrir um arquivo;
  - Atribuição de um valor impossível a um objeto;
  - Divisão por zero;
  - Etc.

Erros de execução (exceção): Exemplo

```
1  package media;
2
3  import java.util.Scanner;
4
5  public class Media {
6
7          public static void main(String[] args) {
8
9              Scanner entrada = new Scanner(System.in);
10
11             float a = entrada.nextFloat();
12             float b = entrada.nextFloat();
13             float c = entrada.nextFloat();
14
15             float media = (a+b+c)/3;
16
17             System.out.println( "A média é : " + media);
18         }
19 }
```

Saída - Media (run)

```
run:
2
a
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:909)
        at java.util.Scanner.next(Scanner.java:1530)
        at java.util.Scanner.nextFloat(Scanner.java:2388)
        at media.Media.main(Media.java:12)
Java Result: 1
CONSTRUÍDO COM SUCESSO (tempo total: 10 segundos)
```

Erros de execução (exceção)

- Os erros durante a execução dos programas podem **causar a interrupção** dos mesmos, com consequências das mais **imprevisíveis**.

- Chamamos de exceção a um problema durante a execução do programa.

- O **tratamento de exceções** é utilizado em situações em que o sistema, no caso da linguagem Java a JVM, pode se **recuperar do mau funcionamento** que causou a exceção.

Tratamento de exceções em Java

- A linguagem Java disponibiliza um mecanismo para tratamento das exceções:

```
try {
    // código a ser executado
} catch (ClasseDeExceção instânciaDaExceção) {
    // tratamento da exceção
} finally {
   // código a ser executado mesmo que uma exceção seja lançada
}
```

Tratamento de exceções em Java

- **try (tentar)** é usada para indicar um bloco de código que possa ocorrer uma exceção.
- **catch (pegar)** serve para manipular as exceções, ou seja, tratar o erro. Pode existir mais de um bloco catch.
- **ClasseDeExceção e instânciaDaExceção** classe da exceção a ser tratada;
- **finally (finalmente)** sempre será executado depois do bloco try/catch e é opcional.

Tratamento de exceções: Exemplo

```
1  package media;
2
3  import java.util.Scanner;
4
5  public class Media {
6
7      public static void main(String[] args) {
8
9          Scanner entrada = new Scanner(System.in);
10
11         try{
12
13          float a = entrada.nextFloat();
14          float b = entrada.nextFloat();
15          float c = entrada.nextFloat();
16          float media = (a+b+c) /3;
17          System.out.println( "A média é : " + media);
18         }
19         catch (Exception erro) {
20          System.out.println("Atenção ! Entrada Inválida !");
21         }
22     }
23 }
```

```
Saída - Media (run)
run:
1
2
3
A média é : 2.0
CONSTRUÍDO COM SUCESSO (tempo total: 4 segundos)
```

```
Saída - Media (run)
run:
1
2
a
Atenção ! Entrada Inválida !
CONSTRUÍDO COM SUCESSO (tempo total: 7 segundos)
```

Tratamento de exceções: Exemplo

- Usando finally

```
1   package media;
2
3   import java.util.Scanner;
4
5   public class Media {
6
7           public static void main(String[] args) {
8
9               Scanner entrada = new Scanner(System.in);
10
11              try{
12
13               float a = entrada.nextFloat();
14               float b = entrada.nextFloat();
15               float c = entrada.nextFloat();
16               float media = (a+b+c) /3;
17               System.out.println( "A média é : " + media);
18              }
19              catch (Exception erro) {
20               System.out.println("Atenção ! Entrada Inválida !");
21              }
22              finally{
23              System.exit(0);
24              }
25          }
26  }
```

Tratamento de exceções - bloco finally

- A função básica de **finally** é sempre executar seu bloco de dados mesmo que uma exceção seja lançada.

- É muito útil para liberar recursos do sistema quando utilizamos, por exemplo, conexões de banco de dados e abertura de buffer para leitura ou escrita de arquivos, etc.

Tratamento de exceções: Exemplo 2

Manipulação de Exceções Comuns

```
try {
    int[] numeros = {1, 2, 3};
    System.out.println("Elemento no índice 3: " + numeros[3]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Erro: Acesso a índice fora dos limites do array.");
} finally {
    System.out.println("Finalizando a execução.");
}
```

Tratamento de exceções: Exemplo 3

Múltiplos Blocos catch

```
try {
    // Código que pode lançar uma exceção
    int resultado = 10 / 0; // Exemplo de divisão por zero
} catch (ArithmeticException e) {
    // Tratamento da exceção específica
    System.out.println("Erro: Divisão por zero não é permitida.");
} catch (Exception e) {
    // Tratamento genérico para outras exceções
    System.out.println("Ocorreu um erro: " + e.getMessage());
} finally {
    // Código que sempre será executado
    System.out.println("Bloco 'finally' executado.");
}
```

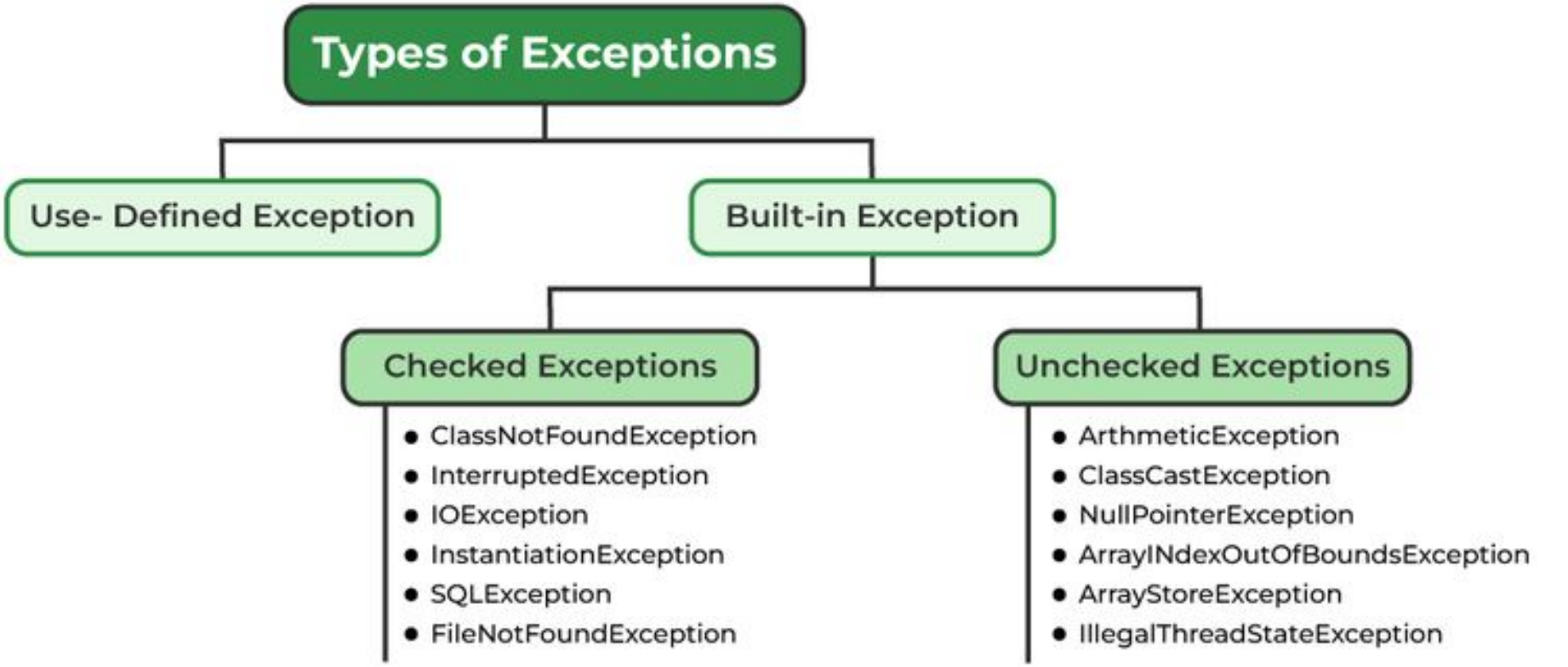
Tratamento de exceções - tipos

- Em Java, as exceções são divididas em duas categorias principais:

- **Checked Exceptions (Exceções Verificadas)**: São verificadas em tempo de compilação.
  - É obrigatório tratá-las usando **try-catch** ou declarando-as com **throws** na assinatura do método.

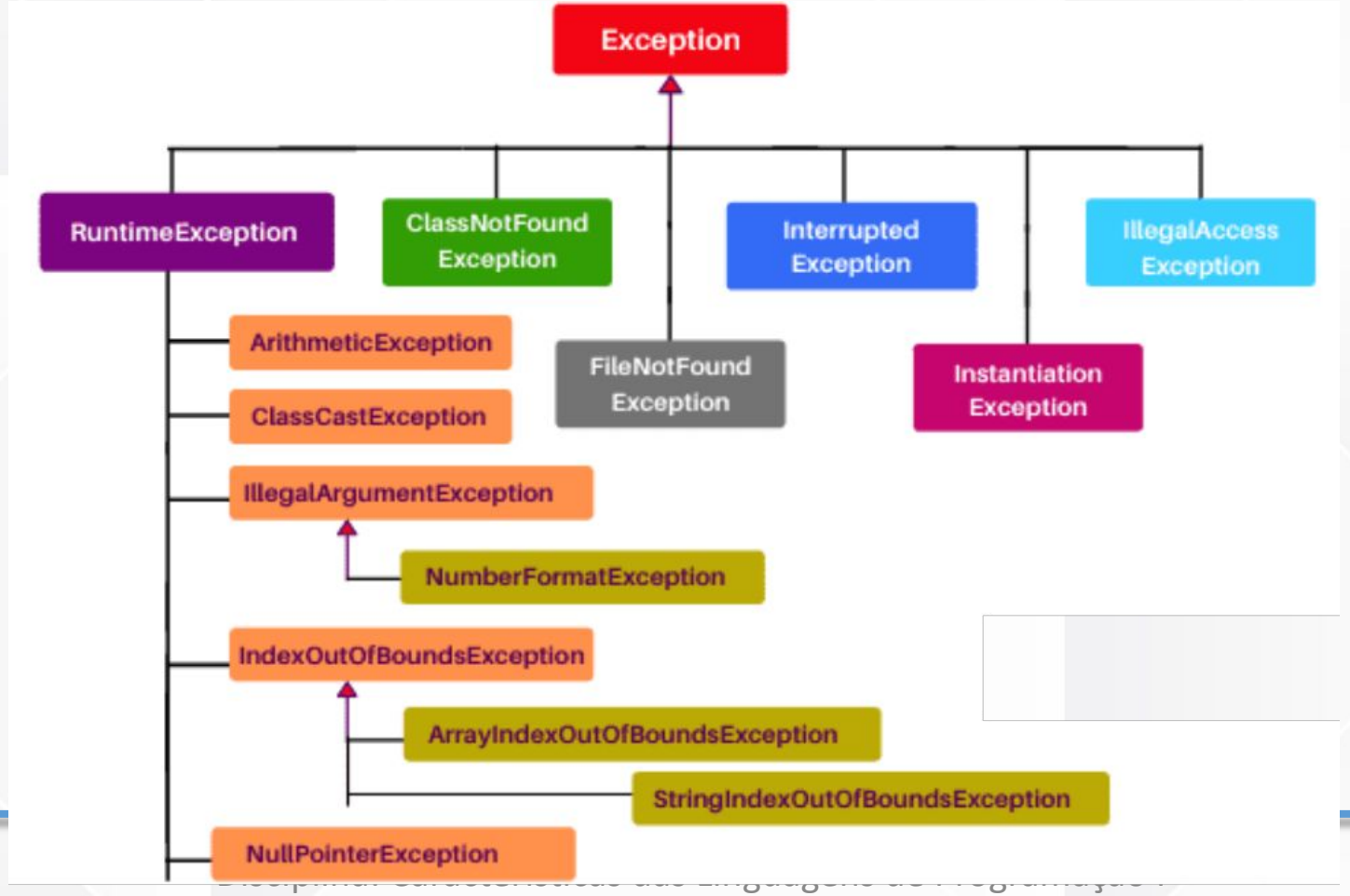
- **Unchecked Exceptions (Exceções Não Verificadas)**: São subclasses de RuntimeException e não são verificadas em tempo de compilação.
  - Elas geralmente indicam erros de lógica do programa, como erros de programação que podem ser evitados.

Tratamento de exceções - Classes



Tratamento de exceções - classe Exception

- Classe base para todas as exceções verificadas.
- Qualquer exceção que não seja uma RuntimeException ou Error é uma subclasse de Exception.



Algumas Exceções verificadas

- **IOException:** Lançada quando ocorre um erro de I/O (entrada/saída), como problemas ao ler ou escrever em um arquivo. Exemplo: Falha ao abrir um arquivo que não existe:

- **FileNotFoundException**: Subclasse de IOException. Lançada quando um arquivo especificado não pode ser encontrado. Exemplo: Tentativa de abrir um arquivo com o caminho incorreto.

- **SQLException**:Relacionada a erros no acesso ao banco de dados usando JDBC. Exemplo: Falha na execução de uma consulta SQL ou problemas de conexão.

- **ClassNotFoundException:** Lançada quando a JVM não consegue encontrar a definição de uma classe no tempo de execução. Exemplo: Tentativa de carregar uma classe que não está presente no classpath.

Algumas Exceções não verificadas

- **RuntimeException**: Superclasse para exceções que podem ocorrer durante a execução do programa e que normalmente indicam erros de programação.
- **NullPointerException** : Lançada quando há uma tentativa de usar um objeto que está null.
- **ArrayIndexOutOfBoundsException**: Lançada quando é feito acesso a um índice fora dos limites de um array.
- **ArithmeticException** : Lançada para erros aritméticos, como divisão por zero.
- **ClassCastException:** Lançada quando há uma tentativa de converter um objeto para uma subclasse da qual ele não é uma instância.
-

Algumas Exceções não verificadas (2)

- **IllegalArgumentException:** Lançada para indicar que um método recebeu um argumento inválido.
- **IndexOutOfBoundsException:** Superclasse para exceções que ocorrem ao acessar índices inválidos em listas, arrays, etc.
- **IllegalStateException**: Lançada quando a invocação de um método é inválida para o estado atual do objeto.
- **UnsupportedOperationException**: Lançada para indicar que a operação solicitada não é suportada.

Forçando exceções: throw

- A palavra-chave **throw** em Java é usada para lançar explicitamente uma exceção.

- É útil quando queremos indicar que ocorreu uma condição de erro específica no código, ou quando queremos sinalizar uma situação excepcional que o código que chamou o método deve tratar.

- Sintaxe:

```
throw new TipoDeExcecao("Mensagem de erro");
```

  Onde:
  - TipoDeExcecao deve ser uma classe que herda de Throwable (geralmente Exception ou RuntimeException).
  - Podemos fornecer uma mensagem que descreve o erro (opcional).

Exemplo de uso de throw

- Lançando uma Exceção Personalizada

```
public static void validarIdade(int idade) {
    if (idade < 18) {
        throw new IllegalArgumentException("Idade deve ser maior ou igual a 18.");
    }
    System.out.println("Idade válida: " + idade);
}
```

```
public static void main(String[] args) {
    try {
        validarIdade(15); // Isto lançará uma exceção
    } catch (IllegalArgumentException e) {
        System.out.println("Erro capturado: " + e.getMessage());
    }
}
```

Exemplo de uso de throw

- Lançando uma Exceção Personalizada com uma classe própria

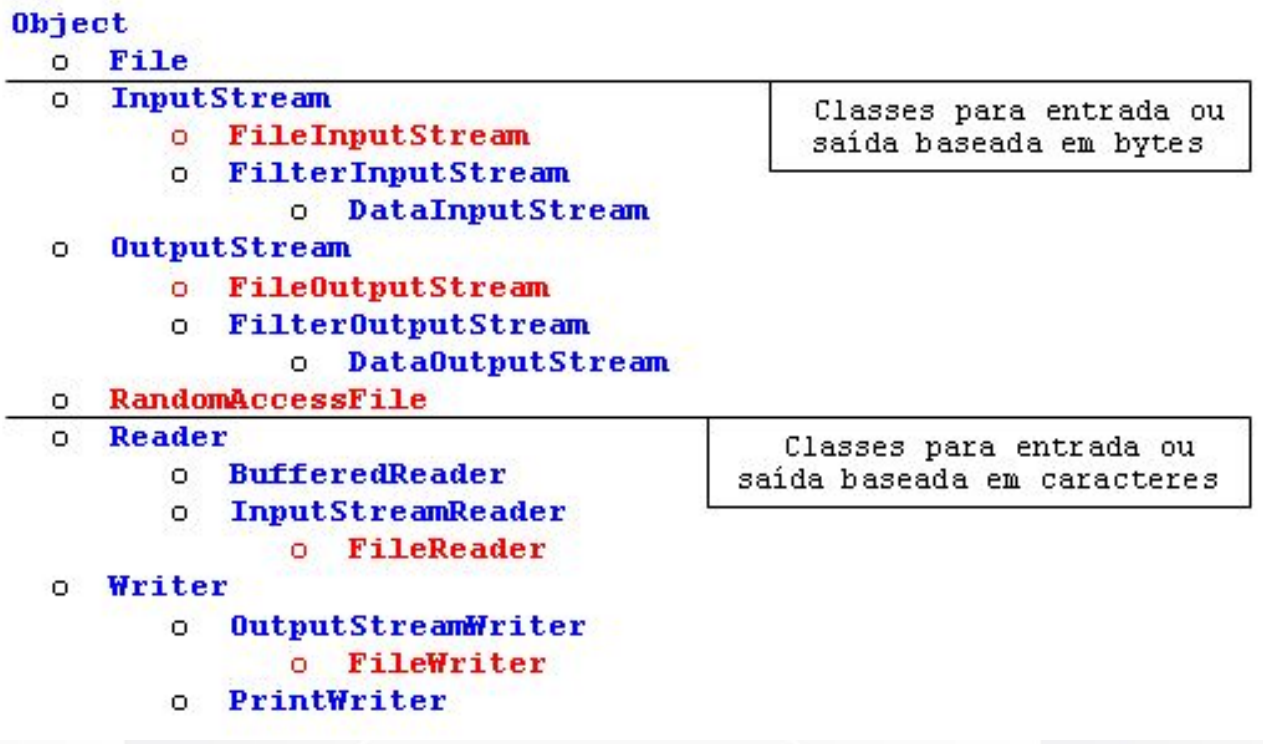
```
class SaldoInsuficienteException extends Exception {
    public SaldoInsuficienteException(String mensagem) {
        super(mensagem);
    }
}
```

```
public class ExemploThrow{
    public static void sacar(double saldo, double quantia) throws SaldoInsuficienteException {
        if (quantia > saldo) {
            throw new SaldoInsuficienteException("Saldo insuficiente!");
        }
        System.out.println("Saque realizado com sucesso!");
    }
    public static void main(String[] args) {
        try {
            sacar(500.0, 700.0); // Tentativa de sacar
        } catch (SaldoInsuficienteException e) {
            System.out.println("Erro: " + e.getMessage());
        }
    }
}
```

Arquivos

Arquivos em Java

- Programas Java implementam o processamento de arquivos utilizando as classes do pacote **java.io**. O pacote oferece mais de 50 classes distintas para o processamento de entrada e saída em arquivos baseados em bytes e caracteres e arquivos de acesso aleatório. Os arquivos são abertos criando-se objetos.
-



Arquivos em Java

Algumas das classes mais utilizadas do pacote java.io para manipulação de arquivos são:

- **FileInputStream**: para entrada baseada em bytes de um arquivo.
- **FileOutputStream**: para saída baseada em bytes para um arquivo.
- **RandomAccessFile**: para entrada e saída baseada em bytes de e para um arquivo.
- **FileReader**: para entrada baseada em caracteres de um arquivo.
- **FileWriter**: para saída baseada em caracteres para um arquivo.

java.io.File

A classe File representa um arquivo ou diretório no sistema operacional. Importante saber que apenas representa, não significa que o arquivo ou diretório realmente exista.

Para instanciar um objeto do tipo File:

```
File arquivo = new File( "c:/nome_do_arquivo.txt" );
```

Com o objeto instanciado, é possível fazer algumas verificações, como por exemplo se o arquivo ou diretório existe:

```
//verifica se o arquivo ou diretório existe
boolean existe = arquivo.exists();
```

Caso não exista, é possível criar um arquivo ou diretório:

```
//cria um arquivo (vazio)
arquivo.createNewFile();

//cria um diretório
arquivo.mkdir();
```

java.io.File

Caso seja um diretório, é possível listar seus arquivos e diretórios através do método listFiles(), que retorna um vetor de File:

```
//caso seja um diretório, é possível listar seus arquivos e diretórios
File [] arquivos = arquivo.listFiles();
```

É possível também excluir o arquivo ou diretório através do método delete(). Uma observação importante é que, caso seja um diretório, para poder excluir, este tem de estar vazio:

```
//exclui o arquivo ou diretório
arquivo.delete();
```

java.io.FileWritter e java.io.BufferedWritter

- As classes **FileWriter** e **BufferedWriter** servem para escrever em arquivos de texto.
- A classe **FileWriter** serve para escrever **diretamente** no arquivo, enquanto a classe **BufferedWriter**, além de ter um desempenho melhor, possui alguns **métodos** que são independentes de **sistema operacional**, como quebra de linhas.
- Para instanciar um objeto do tipo FileWriter:

```
//construtor que recebe o objeto do tipo arquivo
FileWriter fw = new FileWriter( arquivo );

//construtor que recebe também como argumento se o conteúdo será acrescentado
//ao invés de ser substituído (append)
FileWriter fw = new FileWriter( arquivo, true );
```

java.io.FileWritter e java.io.BufferedWritter

- A criação do objeto BufferedWriter

```
//construtor recebe como argumento o objeto do tipo FileWriter
BufferedWriter bw = new BufferedWriter( fw );
```

- Com o bufferedwriter criado, agora é possível escrever conteúdo no arquivo através do método write():

```
//escreve o conteúdo no arquivo
bw.write( "Texto a ser escrito no txt" );

//quebra de linha
bw.newLine();
```

- Após escrever tudo que queria, é necessário fechar os buffers e informar ao sistema que o arquivo não está mais sendo utilizado:

```
//fecha os recursos
bw.close();
fw.close();
```

java.io.FileReader e java.io.BufferedWritter

- As classes **FileReader** e **BufferedReader** servem para ler arquivos em formato texto. A classe FileReader recebe como argumento o objeto File do arquivo a ser lido:

```
//construtor que recebe o objeto do tipo arquivo
FileReader fr = new FileReader( arquivo );
```

- A classe BufferedReader, fornece o método readLine() para leitura do arquivo:

```
//construtor que recebe o objeto do tipo FileReader
BufferedReader br = new BufferedReader( fr );
```

- Para ler o arquivo, basta utilizar o método ready(), que retorna se o arquivo tem mais linhas a ser lido, e o método readLine(), que retorna a linha atual e passa o buffer para a próxima linha:

```
//equanto houver mais linhas
while( br.ready() ){
    //lê a proxima linha
    String linha = br.readLine();
    //faz algo com a linha
}
```

- Da mesma forma que a escrita, a leitura deve fechar os recursos:

```
br.close();
fr.close();
```

Arquivos em Java: exemplo 1

Crie uma nova classe e no método main insira o código abaixo:

```
1  public static void main(String[] args) {
2      File arquivo = new File("/nome_do_arquivo.txt");
3      try {
4          if (!arquivo.exists()) {
5              //cria um arquivo (vazio)
6              arquivo.createNewFile();
7          }
8          //caso seja um diretório, é possível listar seus arquivos e diretórios
9          File[] arquivos = arquivo.listFiles();
10         //escreve no arquivo
11         FileWriter fw = new FileWriter(arquivo, true);
12         BufferedWriter bw = new BufferedWriter(fw);
13         bw.write("Texto a ser escrito no txt");
14         bw.newLine();
15         bw.close();
16         fw.close();
```

Arquivos em Java: exemplo 1 (cont.)

continua…

```
18          //faz a leitura do arquivo
19          FileReader fr = new FileReader(arquivo);
20          BufferedReader br = new BufferedReader(fr);
21          //enquanto houver mais linhas
22          while (br.ready()) {
23              //lê a proxima linha
24              String linha = br.readLine();
25              //faz algo com a linha
26              System.out.println(linha);
27          }
28          br.close();
29          fr.close();
30      } catch (IOException ex) {
31          ex.printStackTrace();
32      }
33  }
```

Arquivos em Java: exemplo 2

Crie a classe Arquivos com dois métodos estáticos para leitura e escrita de arquivos

```
1  import java.io.BufferedReader;
2  import java.io.BufferedWriter;
3  import java.io.FileReader;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.util.Scanner;
7
8  public class Arquivos {
9
10     public static void leitor(String path) throws IOException {
11         BufferedReader buffRead = new BufferedReader(new FileReader(path));
12         String linha = "";
13         while (true) {
14             if (linha != null) {
15                 System.out.println(linha);
16
17             } else
18                 break;
19             linha = buffRead.readLine();
20         }
21         buffRead.close();
22     }
```

Arquivos em Java: exemplo 2 (cont.)

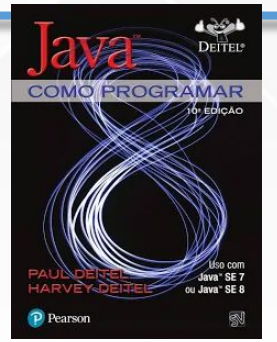
Crie a classe Arquivos com dois métodos estáticos para leitura e escrita de arquivos

```
23
24      public static void escritor(String path) throws IOException {
25          BufferedWriter buffWrite = new BufferedWriter(new FileWriter(path));
26          String linha = "";
27          Scanner in = new Scanner(System.in);
28          System.out.println("Escreva algo: ");
29          linha = in.nextLine();
30          buffWrite.append(linha + "\n");
31          buffWrite.close();
32      }
33
34  }
```

Referências

BIBLIOGRAFIA BÁSICA:

- DEITEL, Harvery M.. Java : como programar. 10ª ed. São Paulo: Pearson - Prentice Hall, 2017.



- BORATTI, Isaías Camilo. Programação Orientada a Objetos em Java : Conceitos Fundamentais de Programação Orientada a Objetos. 1ª ed. Florianópolis: VisualBooks, 2007.

- SIERRA, Kathy; BATES, Bert. Use a Cabeça! Java. 2ª ed. Rio de Janeiro: Alta Books, 2007.