10UNIDADE 2 – ELEMENTOS BÁSICOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

MÓDULO 1 – DEFININDO CLASSES E OBJETOS II

01

1 - A PALAVRA RESERVADA *THIS*

A palavra reservada ***this*** é chamada de ponteiro de autorreferenciação porque ele é usado para referenciar o objeto do método que está sendo chamado.

Para clarear um pouco vamos mostrar um exemplo de sua utilização:

Exemplo 2_1_001: uso da palavra reservada this.

```
class soma{
        private int valor;
        public soma(int num)
        {
               this.valor=num;
        }
        private int getValor()
        {
               return this.valor; //busca o atributo valor
        }
        private int add(int num)
        {
              return (this.Valor+num);
        }
    }
```

Ponteiro

O ponteiro é uma variável que armazena um endereço de memória de outra variável. Por exemplo, a variável A é um ponteiro que contém o endereço de memória da variável B. Dizemos então que a variável A aponta para B ou, ainda, que o ponteiro A referencia B.

A	00000002	00000001
B	‘a’	00000002

02

A palavra **this** também pode ser utilizada para evitar conflitos escopo. Por exemplo, podemos ter os parâmetros de uma função com o mesmo nome do atributo da classe:

Exemplo 2_1_002: uso da palavra reservada this.

```
class soma{
        private int valor;
        public soma(int valor)
        {
                this.valor=valor;
        }
        private int getValor()
        {
                return this.valor; //busca o atributo valor
        }
        private int add(int num)
        {
              return (this.valor+num);
        }
    }
```

03

2 - SOBRECARGA DE CONSTRUTORES E MÉTODOS

Vamos voltar ao exemplo da classe *soma*. Essa classe só aceita somar valores inteiros. Vamos agora fazer algumas alterações usando a propriedade de sobrecarga de construtor para flexibilizar o uso dessa classe.

Exemplo 2_1_003: flexibilizando o uso da classe por meio da sobrecarga do construtor.

```
public class soma2 {
    private double valor;
    public soma2(int valor) /*sobrecarga deconstrutores. Existem dois
                            construtores que diferem pelo tipo do
                                 parâmetro*/
    {
           this.valor=valor;
    }
    public soma2(double valor)
    {
           this.valor=valor;
    }
    private double getValor()
    {
    return this.valor; //busca o atributo valor
    }
    public double add(int num)
    {
         return (this.valor+num);
```

```
    }
    public static void main(String[] args) {
          soma2 val1 = new soma2(12); //usa o primeiro construtor
          soma2 val2 = new soma2(5.5); //usa o segundo construtor
    }
}
```

Vejam, no exemplo anterior, que a classe possui dois construtores. O primeiro será utilizado quando o objeto for criado por meio de um argumento inteiro (linha 22). O segundo construtor será utilizado quando o argumento utilizado para criar o objeto do tipo soma2 for do tipo decimal (linha 23).

No exemplo anterior, temos ainda uma limitação para realizar a soma, pois a função add só recebe argumentos inteiros. A seguir vamos mostrar como tornar isso flexível por meio da sobrecarga de métodos.

04

Exemplo 2_1_004: flexibilizando o uso da classe por meio da sobrecarga de métodos.

```
public class soma3 {
        private double valor;
        public soma3(int valor) /*sobrecarga de construtores. Existem
                                  dois construtores que diferem pelo
                                  tipo do parâmetro*/
        {
              this.valor=valor;
        }
        public soma3(double valor)
        {
              this.valor=valor;
        }
        private double getValor()
        {
        return this.valor; //busca o atributo valor
        }
        public double add(int num)
        {
            this.valor+=valor;
            return (this.valor);
        }
        public double add(double valor)
        {
            this.valor+=valor;
            return (this.valor);
        }
        public static void main(String[] args) {
              soma3 val = new soma3(5.5);
              val.add(3);
```

```
                val.add(4.5);
                System.out.print(val.getValor());
        }
}
```

No exemplo, foi feita a sobrecarga do método add para possibilitar que esse método receba tanto valores inteiros, quanto valores do tipo double. Não há um limite para sobrecargas de construtores e métodos. Podemos fazer tantos quanto forem necessários. O exemplo a seguir ilustra isso.

05

Exemplo 2_1_005: flexibilizando o uso da classe por meio da sobrecarga de métodos.

```
public class soma4 {
        private double valor;
        public soma4(int valor) /*sobrecarga de construtores. Existem
                                        dois construtores que diferem pelo
                                        tipo do parâmetro*/
        {
                this.valor=valor;
        }
        public soma3(double valor)
        {
                this.valor=valor;
        }
        private double getValor()
        {
        return this.valor; //busca o atributo valor
        }
        public double add(int num)
        {
            this.valor+=valor;
            return (this.valor);
        }
        public double add(double valor)
        {
            this.valor+=valor;
            return (this.valor);
        }
        public static void main(String[] args) {
              soma3 val = new soma3(5.5);
              val.add(3);
              val.add(4.5);
              System.out.print(val.getValor());
        }
}
```

A sobrecarga de operadores permite flexibilizar o uso das classes, melhorando a portabilidade e permitindo um melhor reaproveitamento do código.

06

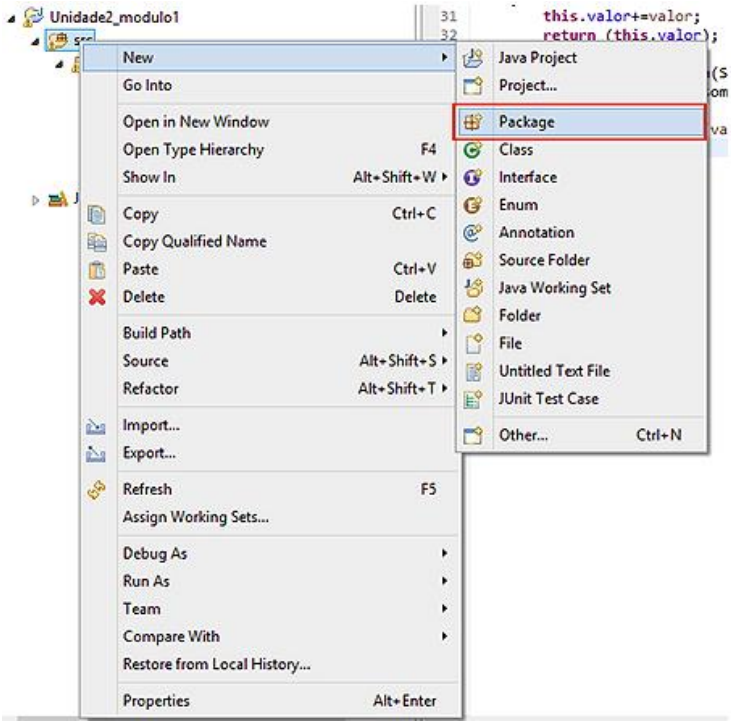
3 - ORGANIZANDO CLASSES – USO DE PACOTES

Quando trabalharmos em um projeto de *software* mais complexo deveremos organizar melhor o projeto em função da grande quantidade de classes que geralmente é necessária para implementação de projetos de médio e grande porte.

A forma mais eficiente de organização de classes é a criação de **pacotes** (“Packages”), também chamadas de **bibliotecas de classes**.

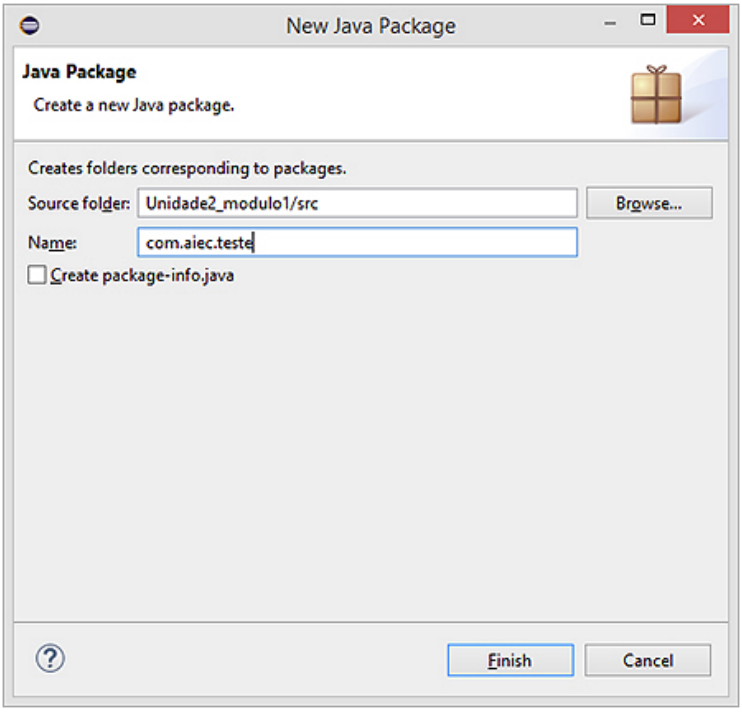
A utilização de pacotes facilita a reutilização do código, e permite o compartilhamento da mesma forma como utilizamos os pacotes padrão do java, por meio do comando *import*.

Até agora usamos em todos os programas o pacote standard (default package). Então iremos criar um novo pacote clicando com o botão direito em cima do diretório src, e selecionar novo pacote:

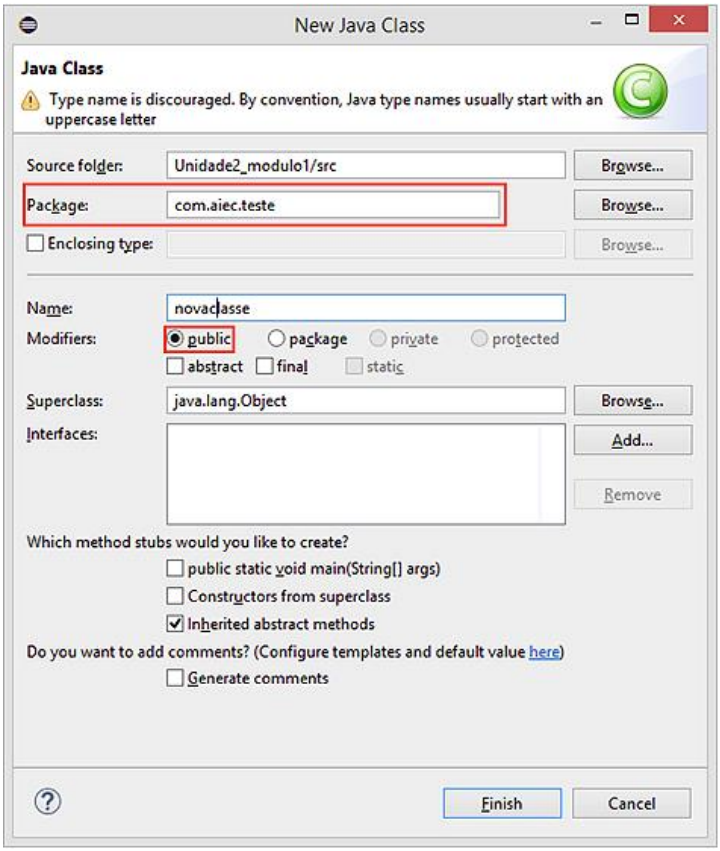


07

Ao abrir a caixa de diálogo, iremos indicar como nome do pacote “com.aiec.teste”.



Agora, podemos incluir uma ou mais classes nesse pacote que acabamos de criar. Para isso basta clicar no pacote e, em seguida, no botão de criação de nova classe C na barra de ferramentas.



Podemos verificar que o próprio Eclipse já preencheu o nome do pacote e fixou a classe como pública para que possa ser acessada por outras classes.

Fechando a caixa de diálogo verificamos que a classe criada já contém a diretiva *package* com.aiec.teste:

```
1  package com.aiec.teste;
2
3  public class novaclasse {
4
5  }
```

08

Para usar essa classe ou qualquer outra classe que venha a ser criada no pacote em qualquer outra classe do projeto basta importar o pacote no arquivo da classe desejada. Por exemplo, incluiremos a esse pacote na classe Soma4:

Exemplo 2_1_006: flexibilizando o uso da classe por meio da sobrecarga de métodos.

```
Import com.aiec.teste.*;

public class soma4 {
        private double valor;
        public soma4(int valor) /*sobrecarga de construtores. Existem
                                    dois construtores que diferem pelo
                                    tipo do parâmetro*/
        {
               this.valor=valor;
        }
        public soma3(double valor)
        {
               this.valor=valor;
        }
        private double getValor()
        {
        return this.valor; //busca o atributo valor
        }
        public double add(int num)
        {
            this.valor+=valor;
            return (this.valor);
        }
        public double add(double valor)
        {
            this.valor+=valor;
            return (this.valor);
        }
```

```
        public static void main(String[] args) {
              soma3 val = new soma3(5.5);
              val.add(3);
              val.add(4.5);
              System.out.print(val.getValor());
        }
}
```

O asterisco depois do nome do pacote significa que estamos importando todas as classes existentes no pacote.

09

4 - DOCUMENTANDO AS SUAS CLASSES- USO DO JAVA DOC

Conforme já vimos na Unidade anterior, podemos comentar trechos do programa em java usando dois tipos de comentário: comentários em linha (//) e multilinha (/**/). Entretanto, o java possui uma forma muito mais poderosa e padronizada de gerar uma documentação reutilizável por outros programadores chamada de javadoc. O javadoc usa comentários multilinhas de forma padronizada para gerar arquivos html que possam servir de documentação online para o código. Mostraremos a seguir um exemplo do código javadoc e o arquivo html que foi gerado:

Exemplo 2_1_007: uso do javadoc.

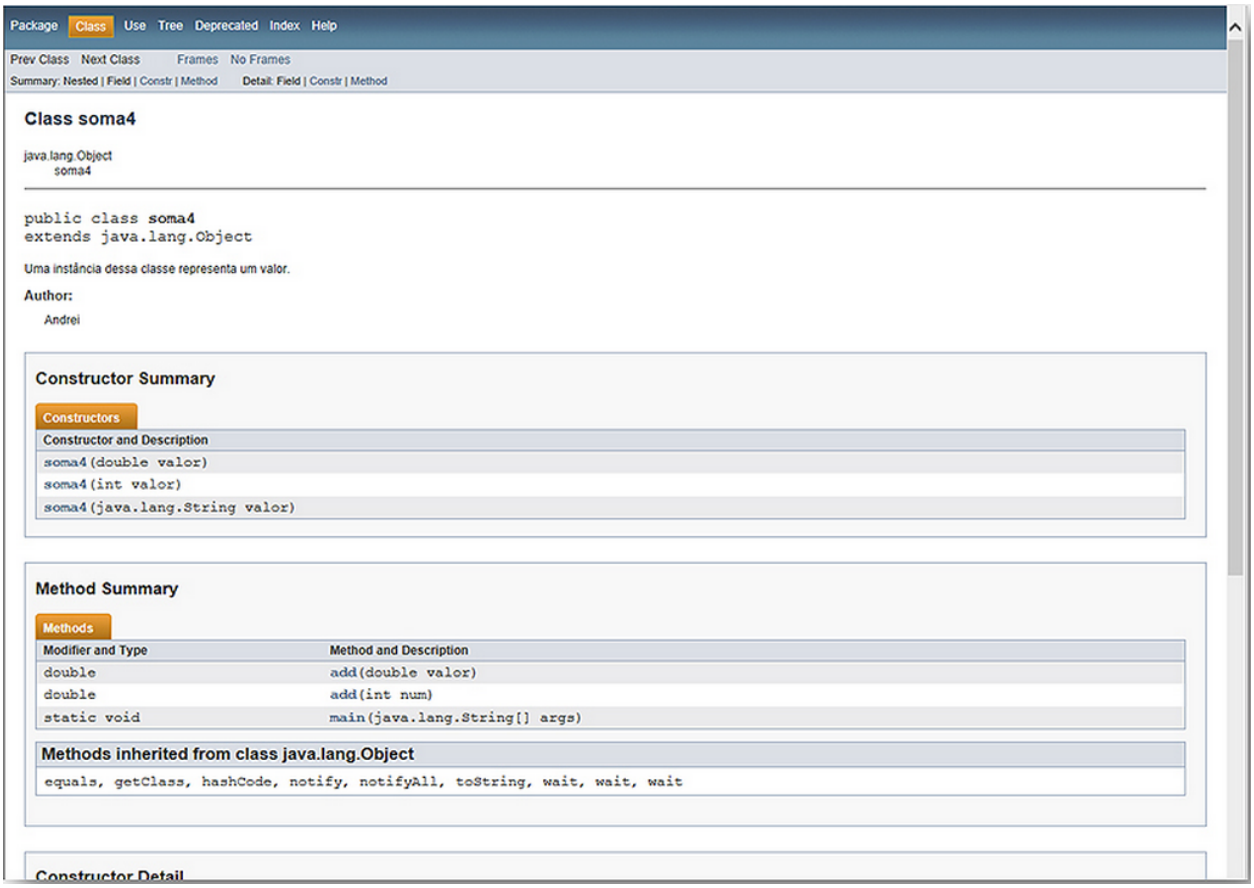
```
import com.aiec.teste.*;
/**
* Uma instância dessa classe representa um valor.
*
* @author Andrei
*
*/
public class soma4 {
    /**
  * The valor em formato double
  */
    private double valor;
    /**
  * The valor em formato texto
  */
    private String texto="";


    public soma4(int valor) /*sobrecarga de construtores. Existem dois
                            construtores que diferem pelo tipo do
                                 parâmetro*/
    {
            this.valor=valor;
    }
```

```
    public soma4(double valor)
    {
           this.valor=valor;
    }
    public soma4(String valor)
    {
           this.texto=valor;
    }
    private double getValor()
    {
    return this.valor; //busca o atributo valor
    }
    /**
  * Calcula a soma do valor atual do objeto
  * com o parâmetro e retorna o resultado.
  *
  * @param num O valor inteiro a ser somado ao objeto
  *
  * @return a soma do valor com o valor atual do objeto
  */
    public double add(int num)
    {
        this.valor+=valor;
        return (this.valor);
    }
    /**
  * Calcula= a soma do valor atual do objeto
  * com o parâmetro do tipo double.
  *
  * @param num O valor do tipo double a ser somado ao objeto
  *
  * @return a soma do valor com o valor atual do objeto
  */
    public double add(double valor)
    {
        this.valor+=valor;
        return (this.valor);
    }
    public static void main(String[] args) {
          soma4 val = new soma4(5.5); //usa o segundo construtor
          val.add(3);
          System.out.print(val.getValor());
    }
}
```

Podemos verificar que ele incluiu os comentários javadoc que descrevem o conteúdo da classe e o autor. E ainda incluiu os construtores e os métodos existentes. Os comentários javadoc começam sempre por **/**** e terminam por ***/**. E são usadas tags especiais começando por arroba @ que definem campos específicos a serem preenchidos.

10

Veja um trecho da documentação gerada pelo javadoc:



A seguir descreveremos as informações necessárias para cada um dos elementos do código.

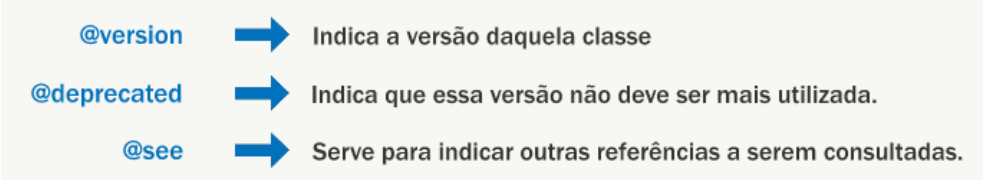
11

- **Documentação da classe**

A documentação da classe tem por finalidade apresentar um mínimo de informação sobre a classe para que possa ser facilmente compreendida e reutilizada. Basicamente consiste em uma descrição e informações sobre o autor da classe. A documentação segue o seguinte formato:

```
/**
* <descrição da classe>
*
* @author <autor_da_classe>
*
*/
```

Além da tag @author também são muito utilizadas para a documentação de classes as seguintes:



Se você deseja aprofundar-se mais e conhecer outras tags, consulte o ***Javadoc Reference Guide***(http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#javadoctags).

12

- **Documentação de um atributo**

A documentação do atributo é feita na linha anterior à sua definição e é feita no seguinte formato:

```
/** <descrição do atributo> */
```

- **Documentação de um método**

A documentação de um método tem por objetivo fornecer uma descrição sucinta, mas suficiente para conhecer o que a função faz, o que ela deve receber como parâmetros e qual seu valor de retorno. Recomenda-se que a descrição comece sempre por um verbo (Busca, Procura, Move, Altera, Edita, etc.) que indique a ação efetuada pelo método. A documentação do método é feita no seguinte formato:

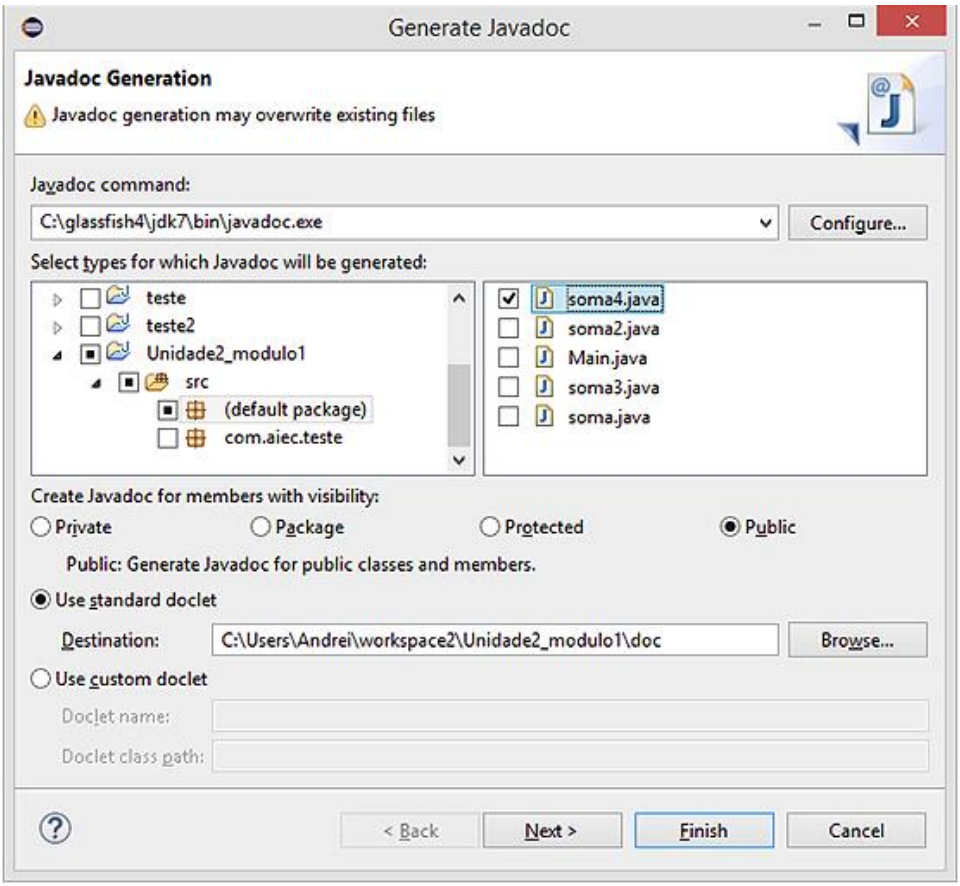
```
/**
* <descrição do método>
*
* @param <nome_do_parâmetro> <descrição>
*
* @return <descrição do valor de retorno>
*/
```

A documentação de métodos construtores é feita da mesma forma que para os demais métodos, entretanto não é utilizada a tag @return, visto que o construtor nunca possui valor de retorno.

13

- **Geração da documentação HTML**

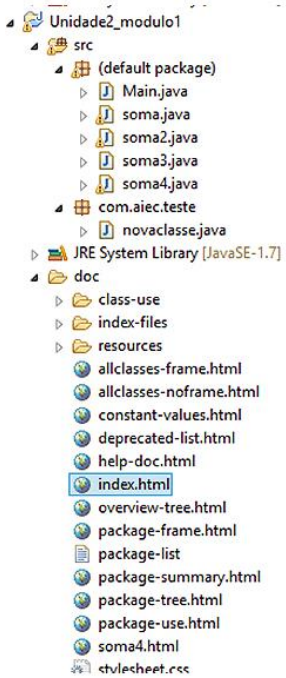
ara gerar a documentação javadoc no Eclipse é simples, basta ir no menu Project->Generate Javadoc:



O campo javadoc command deve ser preenchido com o caminho para o aplicativo javadoc.exe. Caso tenha dúvida, busque no Windows Explorer o caminho do arquivo. Caso ele não se encontre no seu computador, então poderá ser feito o *download aqui*.

14

Uma vez preenchido esse campo, deve-se selecionar a classe a ser documentada e clicar em Finish. Ele irá criar a documentação dentro do projeto em uma pasta chamada Docs:



Para visualizar a documentação, basta dar um clique duplo no arquivo index.html

É importante ressaltar que o uso de javadoc não exclui a utilização de outros comentários ao longo do código. Mesmo que não haja uma intenção de gerar o arquivo HTML com a documentação, o esforço para realizar a documentação é pequeno e já consiste em uma documentação mínima para o projeto que sem dúvida será muito útil facilitar o entendimento do código.

15

RESUMO

Vimos nesse módulo que podemos usar a palavra reservada **this** para referenciar o objeto que está em execução. A palavra reservada pode ser usada para diferenciar um atributo do objeto de um parâmetro da variável que tenha o mesmo nome.

Vimos ainda que a classe pode incluir diversos métodos e que esses métodos podem ter o mesmo nome, desde que a sua assinatura (lista de parâmetros e valor de retorno) seja diferente. Os métodos que possuem o mesmo nome em uma classe são chamados de métodos sobrecarregados (*overloaded*). A sobrecarga também pode ser usada nos métodos construtores, novamente, desde que as assinaturas sejam diferentes.

Mostrou-se que podemos usar pacotes para organizarmos a classe de um projeto o que facilita a manutenção e a reutilização do código.

Por fim, mostramos a utilização do javadoc que possibilita de forma fácil a documentação das classes do projeto. A padronização dos comentários facilita o entendimento de todos os elementos de código e possibilita ainda a geração de uma documentação em formato HTML que poderá ser disponibilizada de

forma online aos demais programadores. Outra vantagem é que o esforço adicional para a inclusão de comentários no padrão javadoc é mínimo não havendo impacto significativo no tempo de desenvolvimento do projeto.

UNIDADE 2 – ELEMENTOS BÁSICOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

MÓDULO 2 – ARRAYS E COLEÇÕES

01

1 - DECLARAÇÃO DE ARRAYS

No dia a dia nos deparamos com uma grande massa de informações. Uma loja de comércio, por exemplo, possui uma grande variedade de produtos que devem ser cadastrados e geridos. Esse mesmo comércio precisa lidar diariamente com clientes que devem ser cadastrados na hora do pagamento e esses mesmos cadastros poderão ser utilizados em uma campanha de marketing. Para armazenar conjuntos grandes de dados do mesmo tipo como os dados dos clientes ou os dados dos produtos as variáveis simples não são apropriadas. Nesse módulo iremos nos aprofundar mais no uso de arrays que foram introduzidos no módulo 4 da unidade 1 e iremos estudar as coleções de dados.

Nós vimos na unidade anterior que os **arrays são uma coleção de dados do mesmo tipo**, sendo declarados conforme abaixo:

```
<tipo_de_dado> [] <nome_do_array>
```

Ou, opcionalmente,

```
<tipo_de_dado> <nome_do_array>[]
```

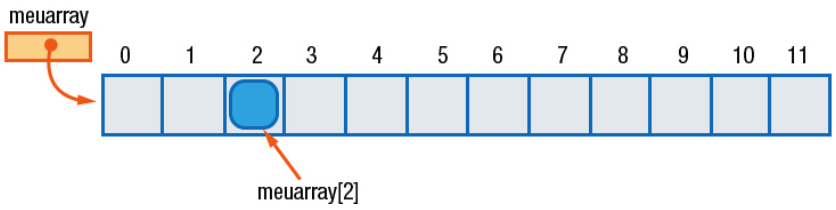
O array é um tipo de dado referenciado, isso significa que a variável declarada do tipo array na verdade apenas armazena o endereço de memória da coleção de dados. Assim, devemos usar o comando new para alocar o espaço necessário e repassar esse endereço para a variável array.

```
<tipo_de_dado> <nome_do_array>[] = new <tipo_de_dado>[<quantidade_de_elementos>]
```

02

No exemplo a seguir criamos uma variável do tipo array que recebe um endereço de memória de um espaço para armazenar 10 valores do tipo double:

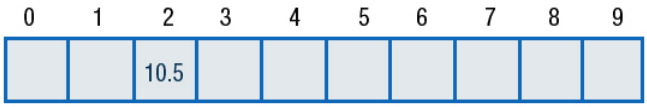
```
double [] meuarray = new double[10];
```



Cada elemento do array é indexado por um índice que começa em zero para o primeiro elemento e vai até n-1, onde n é a quantidade de elementos do array. Para acessar um elemento específico do array usamos o nome do array seguido do índice do elemento entre colchetes:

```
meuarray[2] = 10.5;
```

Teremos:



Também é possível inicializarmos o array no momento da declaração:

```
String[] meses = {“janeiro”,”fevereiro”,”março”,”abril”,”maio”, “junho”,
“julho”,”agosto”,”setembro”,”outubro”, “novembro”,”dezembro”};
```

No caso acima não é necessário especificar a quantidade de elementos, pois o próprio interpretador irá contar a quantidade de elementos que foram usados na inicialização e obter o valor. Caso se queira saber a quantidade de elementos de um array, basta usar o atributo length.

```
int tamanho_do_array = meses.length;
```

03

2 - Arrays de objetos

Os arrays podem servir para armazenar coleções de objetos definidos pelo usuário. Primeiramente, vamos apresentar um programa que define uma classe chamada livro:

Exemplo_2_2_001: classe livro

```
public class livro {

        private String titulo;
        private String autor;
        private int data;
        private String editora;
```

```
        public livro(String titulo, String autor, int data, String editora){
            this.titulo=titulo;
            this.autor=autor;
            this.data=data;
            this.editora=editora;
        }
        public void setTitulo(String titulo){
            this.titulo=titulo;
        }
        public void setAutor(String autor){
            this.autor=autor;
        }
        public void setData(int data){
            this.data=data;
        }
        public void setEditora(String editora){
            this.editora=editora;
        }
        public String getTitulo(){
            return titulo;
        }
        public String getAutor(){
            return autor;
        }
        public int getData(){
            return data;
        }
        public String getEditora(){
            return editora;
        }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        livro livro1 = new livro("Linguagem Java", "Paulo Costa",1998,
"Pearson");
        System.out.println( "Título: " + livro1.getTitulo() );
        System.out.println( "Autor: " + livro1.getAutor() );
        System.out.println( "Data: " + livro1.getData() );
        System.out.println( "Editora: " + livro1.getEditora() );
    }
}
```

04

Vamos agora alterar o programa anterior para que ele possa armazenar dez livros. O primeiro passo é declara um array chamado livros:

```
livro[] livros = new livro[10];
```

Observe que a declaração acima cria apenas o array e reserva espaço para armazenar 10 objetos do tipo livro, entretanto, nenhum dos 10 objetos foi criado. Assim, teremos que criá-los usando o comando new:

```
livros[0] = new livro ("Java Para Iniciantes", "Luis Carlos Moreira da Costa",1998, "Ciencia Moderna");
```

O comando acima cria o objeto livro o aloca no primeiro elemento do array.

Exemplo_2_2_002: exemplo de criação de array de objeto da classe livro

```
public class livro {
        private String titulo;
        private String autor;
        private int data;
        private String editora;

        public livro(String titulo, String autor, int data, String editora){
            this.titulo=titulo;
            this.autor=autor;
            this.data=data;
            this.editora=editora;
        }
        public void setTitulo(String titulo){
            this.titulo=titulo;
        }
        public void setAutor(String autor){
            this.autor=autor;
        }
        public void setData(int data){
            this.data=data;
        }
        public void setEditora(String editora){
            this.editora=editora;
        }
        public String getTitulo(){
            return titulo;
        }
        public String getAutor(){
            return autor;
        }
        public int getData(){
            return data;
        }
        public String getEditora(){
            return editora;
        }
    public static void main(String[] args) {
        //declaração do array com 10 elementos do tipo livro
```

```
    livro[] livros = new livro[10];

        //Declaração de cada um dos elementos.
    livros[0] = new livro ("Java Para Iniciantes", "Luis Carlos Moreira da
Costa",1998, "Ciencia Moderna");
    livros[1] = new livro ("Livro - Programação em Java: Curso Completo",
"Pedro Coelho",2014, "FCA");
    livros[2] = new livro ("Livro - Java Para Iniciantes", "Herbert
Schildt",2013, "Bookman");
    livros[3] = new livro ("Programação com Java: Uma Introdução
Abrangente", "Herbert Schildt/Dale Skrien",2013, "Bookman");
    livros[4] = new livro ("Use a Cabeça! Java", "Kathgy Sierra & Bert
Bates",2007, "Alta Books");
    livros[5] = new livro ("Programação Java para a Web", "Décio Heinzelmann
Luckow",2010, "Novatec");
    livros[6] = new livro ("Programação de Computadores em Java", "Rui Rossi
dos Santos",2014, "Nova Terra");
    livros[7] = new livro ("Java 7 - Ensino Didático", "Sérgio
Furgeri",2010, "Erica");
    livros[8] = new livro ("Java", "Rui Rossis",2013, "Nova Terra");
    livros[9] = new livro ("Java 7", "Sérgio Furgeri",2008, "Erica");

    //Imprime as informações de cada livro cadatrado
    for(int i=0;i<livros.length;i++){
            System.out.println( "Título: " + livros[i].getTitulo() );
            System.out.println( "Autor: " + livros[i].getAutor() );
            System.out.println( "Data: " + livros[i].getData() );
            System.out.println( "Editora: " + livros[i].getEditora() );
            System.out.println( "--------------------------------" );
    }
    }
}
```

O laço *for* ao final do programa imprime na tela todos os livros cadastrados.

05

3 - PASSANDO ARRAYS PARA MÉTODOS

Os métodos podem receber parâmetros do tipo array. A passagem de arrays para métodos é bastante eficiente em java, pois sendo as variáveis do tipo array referenciadas, isso significa que não há nenhuma cópia de dados na memória. O que é passado para o método é o endereço de memória onde os dados do array estão armazenados, tornando esse processo muito mais rápido e eficiente. Para exemplificar, vamos criar no mesmo projeto anterior a classe *biblioteca*, conforme abaixo.

Exemplo_2_2_003: exemplo de passagem de array como parâmetro para a função.

```
/**
```

```
* A instância dessa classe armazena uma coleção de livros
* @autor Andrei
*
*/
public class biblioteca {

    livro[] livros;
    int quantidade;

    /**
   * Método construtor da classe
   *
   * @param array de livros a serem armazenados
   * */
    public biblioteca(livro[] livros){
        this.quantidade=livros.length;
        this.livros = new livro[this.quantidade];
        for(int i=0;i<this.quantidade;i++){
            this.livros[i]=new
livro(livros[i].getTitulo(),livros[i].getAutor(),livros[i].getData(),livros[i
].getEditora());
        }
    }
    /**
   * Imprime todos os livros da biblioteca
   *
   * @param não recebe parâmetros
   * @return não possui valor de retorno
   * */
    public void imprimeTodos(){
        for(int i=0;i<livros.length;i++){
            System.out.println( "Título: " + livros[i].getTitulo() );
            System.out.println( "Autor: " + livros[i].getAutor() );
            System.out.println( "Data: " + livros[i].getData() );
            System.out.println( "Editora: " + livros[i].getEditora() );
            System.out.println( "---------------------------------" );
        }
    }
    public static void main(String[] args) {
        livro[] livros = new livro[10];
        livros[0] = new livro ("Java Para Iniciantes", "Luis Carlos Moreira
da Costa",1998, "Ciencia Moderna");
        livros[1] = new livro ("Livro - Programação em Java: Curso Completo",
"Pedro Coelho",2014, "FCA");
        livros[2] = new livro ("Livro - Java Para Iniciantes", "Herbert
Schildt",2013, "Bookman");
        livros[3] = new livro ("Programação com Java: Uma Introdução
Abrangente", "Herbert Schildt/Dale Skrien",2013, "Bookman");
        livros[4] = new livro ("Use a Cabeça! Java", "Kathgy Sierra & Bert
```

```
Bates",2007, "Alta Books");
        livros[5] = new livro ("Programação Java para a Web", "Décio
Heinzelmann Luckow",2010, "Novatec");
        livros[6] = new livro ("Programação de Computadores em Java", "Rui
Rossi dos Santos",2014, "Nova Terra");
        livros[7] = new livro ("Java 7 - Ensino Didático", "Sérgio
Furgeri",2010, "Erica");
        livros[8] = new livro ("Java", "Rui Rossis",2013, "Nova Terra");
        livros[9] = new livro ("Java 7", "Sérgio Furgeri",2008, "Erica");

        biblioteca biblio = new biblioteca(livros);
        biblio.imprimeTodos();
    }
}
```

No exemplo 2_2_003 o construtor recebe como parâmetro um array de livros que serão armazenados na instância do objeto que chamamos de biblio (linha 59).

06

Uma operação comum em arrays é a busca por um determinado registro. Vamos agora criar um método para retornar um registro de um livro em particular.

Exemplo_2_2_004: exemplo de passagem de array como parâmetro para a função.

```
/**
* A instância dessa classe armazena uma coleção de livros
* @autor Andrei
*
*/
public class biblioteca {

...
    /**
   * Imprime um livro específico biblioteca
   *
   * @param index do livro
   * @return não possui valor de retorno
   * */
    public void imprimeLivro(int i){
            System.out.println( "Título: " + livros[i].getTitulo() );
            System.out.println( "Autor: " + livros[i].getAutor() );
            System.out.println( "Data: " + livros[i].getData() );
            System.out.println( "Editora: " + livros[i].getEditora() );
            System.out.println( "--------------------------------" );
    }
    /**
   * Busca o indice de um livro específico
```

```
    *
    * @param titulo do livro
    * @return indice do livro na biblioteca.
    * */
     public int getLivro(String titulo){
         int indice=-1;
         for(int i=0; i<livros.length;i++){
             if(livros[i].getTitulo().equals(titulo)){
                 return i;
             }
         }
         return indice;
     }

     public static void main(String[] args) {
     ...
         biblioteca biblio = new biblioteca(livros);
         int indice = biblio.getLivro("Livro - Java Para Iniciantes");
         if(indice>0){
             biblio.imprimeLivro(indice);
         }
     }
}
```

Para fazer a comparação do título utilizamos o método equals(). Isso permite usar o mesmo método de comparação para qualquer tipo de dado. Uma observação importante é que estamos comparando nesse caso o título inteiro.

07

4 - Laço For-Each

O laço for-each é uma forma mais prática de fazer um processo de iteração dos elementos de um array. O que esse comando faz é criar uma variável local do mesmo tipo do elemento do array. Essa variável referenciará um elemento do array a cada iteração até que todos os elementos tenham sido referenciados.

A sintaxe do comando é a seguinte:

```
for ( <type> <variable> : <array> )
        <loop body>
```

Alteramos o método imprimeTodos() de forma a utilizar o laço **for-each**:

```
public void imprimeTodos(){
        for(livro book:livros){
            System.out.println("Título: " + book.getTitulo() );
```

```
            System.out.println("Autor: " + book.getAutor() );
            System.out.println("Data: " + book.getData() );
            System.out.println("Editora: " + book.getEditora() );
            System.out.println("---------------------------------" );
        }
    }
```

08

A variável local book na primeira iteração referenciará o elemento livros[0]. Na próxima iteração referenciará o elemento livros[1] e assim por diante até chegar ao último elemento do array.

Um ponto importante do laço for-each é que os elementos do array poderão ser acessados, mas não poderão ser alterados. Isso significa que não teria sentido tentar anular os objetos de uma array fazendo o seguinte:

```
for(livro book:livros){
          book = null;
    }
```

Entretanto, é possível alterar o conteúdo do objeto. Por exemplo, poderíamos usar o método setTitulo() dentro do laço for-each para apagar o título de todos os elementos do array.

```
public void imprimeTodos(){
        for(livro book:livros){
            book.setTitulo("");
        }
    }
```

09

5 - ARRAYS MULTIDIMENSIONAIS

A planilha que armazena as notas de uma turma de alunos contém uma linha para cada aluno e diversas colunas de acordo com a quantidade de pontos de verificação:

Aluno	Nota 1	Nota 2	Nota 3	Nota Final
André	5,0	4,0	7,0	5,3
Carlos	9,0	5,0	6,0	6,7
Maria	8,0	7,0	8,0	7,7
Roberta	8,0	4,0	5,0	5,7
Sérgio	10,0	9,0	10,0	9,7

Esse é um exemplo de uma array bidimensional. Exemplos que utilizam arrays bidimensionais ou multidimensionais são comuns. Felizmente, o java trata de forma simples a criação e o uso de variáveis do tipo array multidimensional. A sintaxe da declaração é feita como segue:

```
<tipo>[][]..[] <nome_do_array>
```

Para cada dimensão deverá ser acrescentado um conjunto de colchetes [] na declaração:

Array simples	int[] valores
Array bidimensional	int[][] valores
Array tridimensional	int[][][] valores

...

Para alocar espaço para o array seguimos o mesmo raciocínio:

Array simples	int[] valores = new int[10]
Array bidimensional	int[][] valores = new int[10][5]
Array tridimensional	int[][][] valores = new int[10][5][3]

...

10

Embora possamos trabalhar com arrays com mais de 2 dimensões, por uma questão de complexidade, geralmente nos limitamos a trabalhar com, no máximo, duas dimensões. Em um array bidimensional, o primeiro índice do elemento é sempre a linha e o segundo a coluna:

```
double[][] Nota= new double[5][4];
      Nota[2][1] = 7;
```

	0	1	2	3
0	5,0	4,0	7,0	5,3
1	9,0	5,0	6,0	6,7
2	8,0	7,0	8,0	7,7
3	8,0	4,0	5,0	5,7
4	10,0	9,0	10,0	9,7

A inicialização de um array bidimensional seria a seguinte:

```
double[][] Nota = {       {5, 4, 7, 5.3},

      {9, 5, 6, 6.7},
```

```
    {8, 7, 8, 7.7},
    {8, 4, 5, 5.7},
    {10, 9, 10, 9.7}    };
```

11

6 - LISTAS

Vimos nos itens anteriores que é muito útil conseguirmos armazenar coleções de dados. Os arrays permitem o armazenamento de coleção de dados, mas a quantidade de dados armazenados é fixa. Ou seja, devemos saber de antemão quantos elementos deveremos armazenar, pois não é possível alterar o tamanho de um array após a sua criação. Sem dúvida, uma saída seria declararmos sempre uma quantidade grande de elementos para não corrermos o risco de não termos espaço (*overflow*), mas o grande problema seria a alocação desnecessária de espaço de memória.

O java possui em sua biblioteca standard java.util várias classes e interfaces que permitem manter uma coleção de objetos de forma eficiente e expansível. Essas classes são chamadas de JCF, ou *Java Collections Framework*.

O java possui uma interface chamada **List** que possui as características necessárias para o armazenamento de coleções de dados. Podemos usar a interface para declarar uma variável que receba qualquer objeto que implemente a interface **List**:

```
List minhaLista;
```

Entretanto, a declaração abaixo estaria errada:

```
List minhaLista = new List();     //Essa declaração está errada!!!!
```

A declaração acima está errada porque List não é uma classe, então não tem como criar um objeto do tipo List. No java existem basicamente duas classes que implementam a interface **List** que são a **LinkedList** e a **ArrayList**. Visto que as duas implementam a interface **List**, poderemos usar as duas da mesma forma, apesar da implementação interna de cada uma delas seja diferente. Assim poderemos declarar um objeto que armazene coleções das seguintes formas:

//criando diretamente um objeto do tipo específico

```
ArrayList minhaColecao = new ArrayList();

LinkedList minhaColecao = new LinkedList();
```

Ou utilizando a interface List:

```
List minhaColecao = new ArrayList();
```

```
List minhaColecao = new LinkedList();
```

12

Quando usamos a interface **List** na declaração estamos dizendo que estamos criando uma variável que irá referenciar qualquer objeto que implemente a interface **List**. Essa flexibilidade é bem interessante quando criamos métodos. Por exemplo, seja a declaração abaixo:

```
Public void imprimeLista(ArrayList array){
        …
        }
```

O método só aceitará um argumento do tipo ArrayList. Entretanto, como sabemos que qualquer lista que implemente o método **List** se comporta da mesma forma, poderíamos criar um método muito mais flexível utilizando a interface:

```
Public void imprimeLista(List array){
        …
        }
```

Agora esse método aceitará qualquer tipo de objeto que implemente a interface ***List***, isto significa que podemos passar um objeto do tipo**ArrayList** ou **LinkedList**, ou qualquer outro objeto que implemente essa interface.

13

a) Declaração de uma Lista

A priori, uma lista pode conter tipos diferentes de dados, ou seja, nada impediria de criar uma lista que tivesse uma **String** no primeiro elemento e um objeto livro no segundo. Entretanto, a boa prática de programação recomenda a utilização de listas homogêneas, formadas por um mesmo tipo de elemento.

A sintaxe de declaração de uma lista é a seguinte:

```
List <<tipo_do_elemento>> <nome_da_lista> = new <tipo_da_lista><<tipo_do_elemento>>();
```

O tipo da lista será, geralmente, ArrayList ou LinkedList. O tipo do elemento pode tanto ser um tipo básico como inteiros (int), texto (String) como qualquer outra classe.

A seguir apresentamos alguns exemplos de declaração de lista.

```
List<int> valores = new ArrayList<int>();    //lista de valores inteiros

List<livro> biblioteca = new ArrayList<livro>();
```

```
List<String> val = new LinkedList<String>();
```

14

b) Para adicionar um elemento para a lista – Método Add()

Abaixo alguns exemplos de inclusão de elementos em uma lista:

```
List<int> valores = new ArrayList<int>();

valores.add(10);

List<livro> livros = new ArrayList<livro>();

livro book = new livro ("Java Para Iniciantes", "Luis Carlos Moreira da Costa",1998, "Ciencia Moderna");

livros.add(book);
```

Para encontrar o tamanho de uma lista usamos o método size. O comando abaixo iria retornar 1:

```
livros.size();
```

15

c) Acessando os elementos de uma lista – Método get(int índice)

Para acessar um elemento específico da lista usamos o método get que deve receber o índice do elemento que estamos buscando. O laço abaixo varreria todos os elementos da lista livros e imprimiria o título:

```
for(int i=0;i<livros.size();i++){

System.out.println(Livros.get(i).getTitulo());
}
```

Uma outra forma de passer por todos os elementos de uma lista seria usando o commando for-each:

```
for(livro book:livros){

System.out.println(Livros.get(i).getTitulo());
}
```

Para acessar todos os elementos de uma lista também podemos lançar mão de um método chamado iterator(). Esse método retorna um objeto do tipo iterator que aponta para o primeiro elemento da lista.

Em seguida é possível incrementar esse ponteiro para que ele passe para o próximo elemento usando o método next() e, verificar se existe o elemento seguinte usando hasNext(), conforme exemplo abaixo:

```
livro book;
iterator<livro> it = livros.iterator();
while(it.hasNext()){
       book = it.next();
       System.out.println(book.getTitulo());

}
```

16

d) Remover elemento da lista – método remove(int índice)

Para remover elementos de uma lista usamos o método remove:

```
livro book = livros.get(1);
Livros.remove(book);
```

Lembre-se de que os laços de iteração como o for-each não permitem alterar o objeto, logo o uso de remove não é permitido dentro desses laços.

É importante observar que todos os elementos de um lista qualquer podem ser removidos pela chamada sucessiva ao método remove(x). Além disso, caso se deseje remover todos os elementos de uma lista com a chamada a um único método, existe disponível também o método clear( ).

Cuidado que ao remover todos os elementos de uma lista tornando-a vazia, o acesso indevido a lista pode ocasionar o lançamento de exceção como por exemplo:

```
...
livros.clear();
livros.get(0);
...
```

A chamada ao método get(0) após a lista se tornado vazia irá lançar uma exceção do tipo "java.lang.IndexOutOfBoundsException"

17

7 - MAPA

O java possui uma outra interface muito usada chamada Map. Basicamente, o Map é uma interface que possibilita implementar classes de lista que podem usar qualquer tipo de objeto como índice, e não apenas valores inteiros como no caso da interface List. Um tipo de uso comum do Map é em lista de palavras e seus significados.

O map define que as entradas possuem duas partes: o valor e a chave. A chave será usada como índice e por isso mesmo não pode ser repetida.

Temos duas classes básicas que implementam a interface Map são elas: **HashMap** e **TreeMap**. Da mesma forma que fizemos com a interface List, preferencialmente, faremos a declaração de objetos usando a interface:

```
Map<<tipo_da_chave>,<tipo_do_valor>> <nome_da_lista>; <nome_da_lista> = new
<tipo_do_mapa><<tipo_da_chave>,<tipo_do_valor>> ();
```

Exemplo:

```
Map<String,livro> livros = new TreeMap<String,livro>();
```

Nesse exemplo usamos uma chave do tipo String para o mapeamento, poderia ser o número ISBN do livro.

18

Os métodos básicos da interface Map são os seguintes:

- **Put**(<chave>,<valor>) – Inclusão de uma nova entrada no map.

```
livro book = new livro ("Java Para Iniciantes", "Luis Carlos Moreira da
Costa",1998, "Ciencia Moderna"); livros.put("isbn 8573931892",book);
```

- **remove**(<chave>) – para remover a entrada com a respectiva chave.

```
livros.remove("isbn 8573931892");
```

- **clear**() – remove todos os elementos da lista.

- **get**(<chave>) – retorna o valor do elemento com a referida chave.

- **containsKey**(<chave>) – verifica se a lista possui elemento com a chave especificada. Se tiver a função retorna verdadeiro. Caso não encontre, o método retorna nulo.

```
boolean resultado = livros.containsKey("isbn 8573931892");
```

- **entrySet() –** esse método retorna o conjunto de entradas da lista. O seu uso é obrigatório quando usamos o comando for-each:

```
for(Map.Entry<String><livro> entry: livros.entrySet()){
    System.out.println(entry.getKey() + ":\n" +entry.getValue() + "\n");
}
```

19

RESUMO

Usualmente nos deparamos com a necessidade de manipularmos coleções de dados. A forma mais simples de fazê-lo em java é por meio de arrays. Os arrays podem ser multidimensionais, mas geralmente usamos arrays unidimensionais e bidimensionais que são mais simples de serem manipulados.

Os elementos do array são sempre do mesmo tipo e o seu tamanho é definido no momento da alocação de memória. Uma vez definido, o tamanho do array não poderá ser mais alterado. O fato de ter um tamanho fixo e de não disponibilizar métodos específicos para as manipulações usuais como remoção de elementos do array restringe o seu uso. Uma forma mais flexível é usar as classes que implementam a interface List.

A interface é um conjunto de critérios e comportamentos que uma determinada classe deve possuir. A interface List define o comportamento (métodos) que uma classe que manipule coleções de objeto deve ter. As duas classes básicas que implementam a interface List são a ArrayList e a LinkedList, sendo a mais usada a ArrayList. Não entramos em detalhes sobre as diferenças específicas dessas duas classes, o que nos importa é que elas implementam a interface List e, portanto, podem ser usadas da mesma forma.

A interface List define que todos os elementos da lista são identificados por um índice inteiro e também os métodos:

- add() para adicionar um elemento na lista
- remove(int índice) para retirar o elemento da lista que ocupa a posição definida pelo índice.
- get(int índice): retorna o elemento da lista que ocupa a posição definida pelo índice.

Algumas vezes estamos preocupados na recuperação da informação e os índices inteiros definidos pela interface List podem não ser os mais indicados ou fáceis de lidar. Sendo assim temos uma outra interface chamada Map. Essa interface permite a criação de listas key-value, ou seja, lista em que cada elemento está associado a uma chave que pode ser de qualquer tipo. A interface define os seguintes métodos básicos:

- put(<chave>,<valor>) – Incluir uma nova entrada na lista.
- remove(<chave>) – remover a entrada com a respectiva chave.
- clear() – remove todos os elementos da lista.
- get(<chave>) – retorna o valor do elemento com a referida chave.
- containsKey(<chave>) – verifica se a lista possui elemento com a chave especificada. Se tiver a função retorna verdadeiro. Caso não encontre, o método retorna nulo.
- entrySet() – esse método retorna o conjunto de entradas da lista.

Agora podemos seguir em frente, certos de que temos o conhecimento necessário para manipular qualquer coleção de objetos de forma eficiente.

UNIDADE 2 – ELEMENTOS BÁSICOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

MÓDULO 3 – PESQUISA E ORDENAÇÃO

01

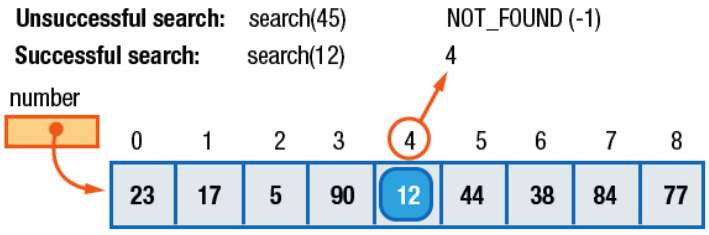
1 - PESQUISA

Um método de pesquisa retorna a posição de um registro da lista dado seu valor. Caso esse valor não esteja presente na lista, o método retornará -1. Existem diversos algoritmos de busca, alguns deles são excelentes para uma busca de sucesso, mas quando o valor procurado não pertence à lista demorará muito mais que outros. Então é bom conhecermos que tipo de buscas são realizadas para determinarmos o melhor algoritmo de pesquisa. Iremos apresentar os algoritmos mais usados.

a) Pesquisa Linear ou sequencial

No método de pesquisa linear o algoritmo receberá um valor a ser procurado e irá compará-lo com cada elemento da lista do início até o fim.

Nesse método o tempo para achar um elemento com o valor procurado é variável, entretanto o tempo necessário para determinar que o elemento não existe na lista é sempre constante, pois corresponde ao tempo necessário para varrer toda a lista.



02

Apresentamos a seguir um exemplo de implementação de pesquisa linear:

Exemplo 1_3_1: exemplo de busca.

```
1   public static void main(String[] args) {
2               // TODO Auto-generated method stub
3               bool achei=false;
4               int[] valores = {4,6,7,8,3,9,2,1,5};
5
6               int valorprocurado = 7;
7
```

```
8               for(int i=0;i<valores.length;i++){
9                    if(valorprocurado==valores[i]){
10                        System.out.println("indice do valor
11  procurado "+ i);
12  Achei=true;
13                   }
                 }
               if(achei) System.out.print("-1");
          }
```

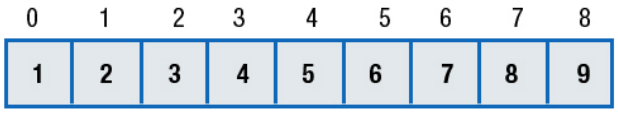
O exemplo apenas imprime o valor do índice procurado. Caso o valor não seja encontrado, será impresso o valor -1. Observe, que no pior caso, teremos que fazer N buscas para obter uma resposta do método.

03

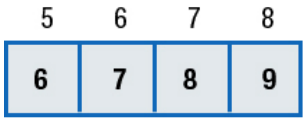
b) Pesquisa Binária

A pesquisa binária é uma forma mais eficiente de efetuarmos a pesquisa, entretanto tem como pré-requisito a necessidade de a **lista estar ordenada**.

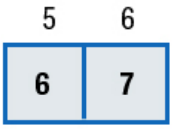
O conceito da pesquisa binária é simples. Vamos usar a lista de valores do exemplo anterior ordenada e buscaremos o valor 6:



Tomando por base a lista ordenada, o primeiro passo da busca binária será comparar o valor procurado com aquele da posição central da lista ordenada. Como o valor procurado 7 é maior que 5, então sabemos que o valor procurado está necessariamente à direita do valor central:



Iremos novamente buscar o elemento central. Como temos um número par de elementos, então iremos comparar o valor procurado com o valor central à esquerda que é 7. Como 7 é maior que o valor procurado, então sabemos agora que o valor procurado está à esquerda do elemento central:



Novamente temos um número par de células, e o elemento que iremos comparar com o valor procurado será o elemento central da esquerda, ou seja, o valor 6. Como o valor é igual então poderemos terminar nossa busca por aqui. Se o valor não fosse igual, poderíamos concluir que o valor não existe na lista.

04

Esse método se chama de **busca binária** porque a cada iteração dividimos o conjunto em dois subconjuntos, até que chegamos a um conjunto com apenas um elemento. Matematicamente podemos escrever que a quantidade de iterações necessárias para buscar um valor usando o método binário é:

$$k=log2N$$

Vamos considerar uma lista com 1024 elementos. Se usássemos o algoritmo de pesquisa linear teríamos que realizar 1024 comparações. Se utilizarmos o método binário, então teremos:

$$k=log21024=10$$

Esse método então melhoraria a eficiência da busca em mais de cem vezes. Vamos agora ver um exemplo de implementação do método de busca binária.

Exemplo 1_3_2: exemplo de busca binária.

```
public class BuscaBinária {


    private static final int NOT_FOUND = -1;
    public static void main(String[] args) {

        int[] valores = {1,2,7,15,35,41,42,48,49,50,51,58,59,60,63,65};
        int retorno = BuscaBinaria(valores,60);
        System.out.println("indice final: "+ retorno);
    }
    public static  int BuscaBinaria(int[] valores, int valorprocurado){
        int inferior, superior,media;

        inferior=0;
        superior = valores.length-1;
        media = (inferior+superior)/2;
        System.out.println("inferior|-------media -------|superior");
        System.out.println("   "+inferior+"\t|-------  "+media+"   -------
|"+superior);
        while(inferior<=superior && valores[media]!=valorprocurado){

            if(valores[media]<valorprocurado){
                inferior=media-1;
            }
```

```
            else{
                superior=media-1;
            }
            media = (inferior+superior)/2;
            System.out.println("   "+inferior+"\t|-------  "+media+"  ------
-|"+superior);
        }
        if(inferior>superior){
            media=-1;
        }
        return media;
    }
}
```

O resultado da execução desse programa será o seguinte:

```
Inferior|-------media -------|superior
0       |-------  7    -------|15
6       |-------  10   -------|15
9       |-------  12   -------|15
11      |-------  13   -------|15
índice final: 13
```

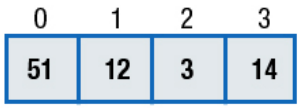
Podemos verificar no resultado que o limite inferior vai sendo alterado até que o valor que ocupa a posição definida pela média se torna igual ao valor procurador.

05

2 - Ordenação (sorting)

Ordenar significa: dada uma lista de valores, colocá-los em ordem crescente ou decrescente.

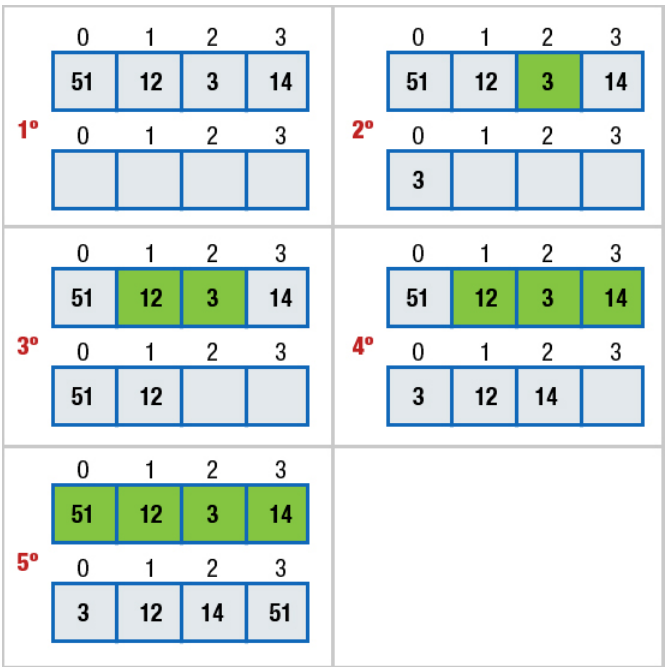
Seja o conjunto de valores abaixo:



Uma forma simples de fazer a ordenação seria criar uma nova lista e seguir o seguinte algoritmo:

1. buscar o menor valor;
2. copiá-lo na lista ordenada na primeira posição vazia;
3. apagá-lo da lista original.
4. Repetir os passos 1 a 3 até que não existam mais valores na tabela original.

Podemos ver no diagrama abaixo o que aconteceria em cada iteração desse algoritmo:

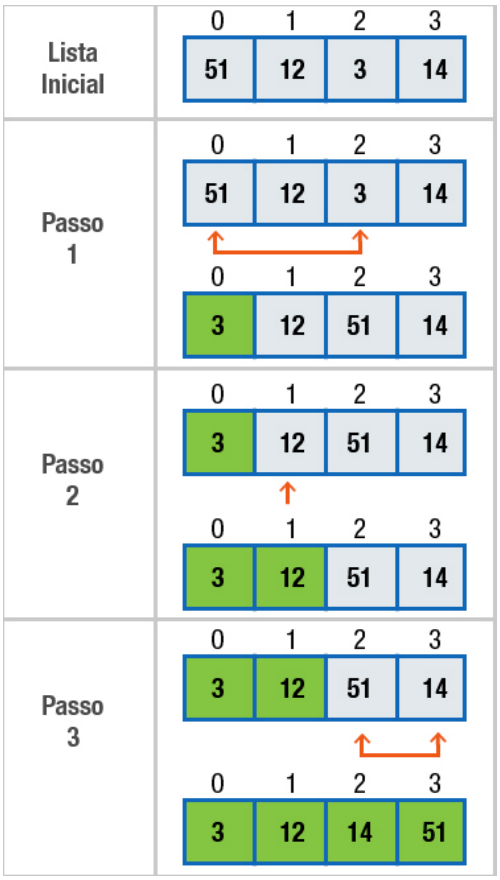


06

Entretanto, podemos ver que esse não é um método muito eficiente, pois em cada iteração ordenamos apenas um elemento do conjunto original, além de usar memória adicional com a cópia ordenada. Um método mais eficiente usa os mesmos princípios desse ordenamento, mas sem a utilização de uma lista auxiliar, pois apenas altera os elementos de posição. O algoritmo é descrito a seguir:

1. buscar o menor valor dentre os elementos não ordenados.
2. trocar esse elemento de posição com o elemento da primeira posição não ordenada.
3. Repetir os passos 1 a 2 até que não haja elementos não ordenados.

Vamos retomar o exemplo anterior e executar o algoritmo:



Esse tipo de método é melhor que o anterior, mas ele não é muito eficiente. A cada iteração movimentamos apenas um elemento, mas fazemos N-i comparações por iteração para saber qual é o menor valor, onde N é a quantidade de elementos e i é o número da iteração. Matematicamente podemos comprovar que a quantidade total de comparações é de N^2. Isso significa que o esforço computacional cresce quadraticamente com a quantidade de elementos a serem ordenados. Veremos um método mais eficiente.

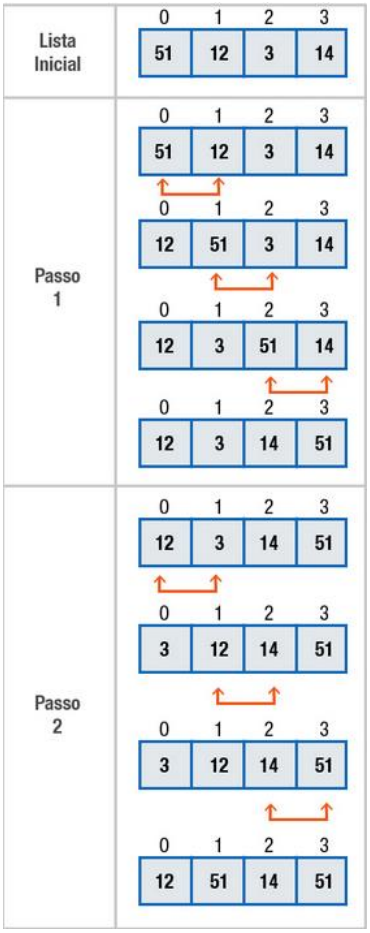
07

a) Método da Bolha (Bubble Sort)

Esse método consiste em comparar e ordenar os elementos do conjunto aos pares. O algoritmo é descrito da seguinte forma:

1. Comparar o elemento com o elemento seguinte, se o elemento seguinte for menor, trocá-los de posição.
2. Repetir o passo 1 para todos os elementos do conjunto.
3. Repetir N/2 vezes os passos 1 e 2.

Novamente, vamos utilizar a lista utilizada anteriormente e aplicar o método da bolha para realizar a ordenação:



Podemos verificar que já na primeira iteração, o maior valor (51) já é colocado na última posição da lista.

08

Vejamos agora um exemplo de implementação desse método de ordenação em java:

Exemplo 1_3_4: exemplo de busca bubbleSort.

```
public class BubbleSort {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int[] valores = {1,2,7,15,35,41,42,48,49,50,51,58,59,60,63,65};
        bubbleSort(valores);
        for(int i=0; i<valores.length;i++){
            System.out.println(valores[i]);
        }
```

```
    }
    public static void bubbleSort(int[] number) {
        int temp, bottom;
        boolean exchanged = true;
        bottom = number.length - 2;
        while (exchanged) {
            exchanged = false;
            for (int i = 0; i <= bottom; i++) {
                if (number[i] > number[i+1]) {
                    temp = number[i]; //exchange
                    number[i] = number[i+1];
                    number[i+1] = temp;
                    exchanged = true; //exchange is made
                }
            }
//        assert maxBottom(number, bottom):
//        "Error: " + number[bottom] +
//        " at position " + bottom +
//        " is not the largest.";
//        bottom--;
        }
//        assert isSorted(number):
//        "Error: the final is not sorted";
    }
    private static boolean maxBottom(int[] number, int lastIndex) {
        for (int i = 0; i < lastIndex; i++) {
            if (number[lastIndex] < number[i]) {
                return false;
            }
        }
        return true;
    }
    private boolean isSorted(int[] number) {
        for (int i = 0; i < number.length-1; i++) {
                if (number[i] > number[i+1]) {
                    return false;
                }
        }
        return true;
    }
}
```

Esse método de pesquisa binária também fará aproximadamente N2 comparações. O método a seguir é mais eficiente.

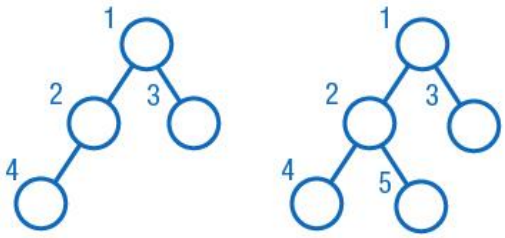
09

3 - HEAPSORTING

É uma outra técnica de ordenação de elementos que é bastante eficiente.

A técnica utiliza uma representação de dados em forma de uma árvore binária, chamada de Heap. Cada elemento do heap recebe uma numeração de 1 até N-1 e recebe um valor. A regra de preenchimento do heap é que todos os níveis devem ser preenchidos antes de passar ao nível de baixo. O nível de baixo é sempre preenchido da esquerda para a direita.

Abaixo apresentamos alguns exemplos de heap:

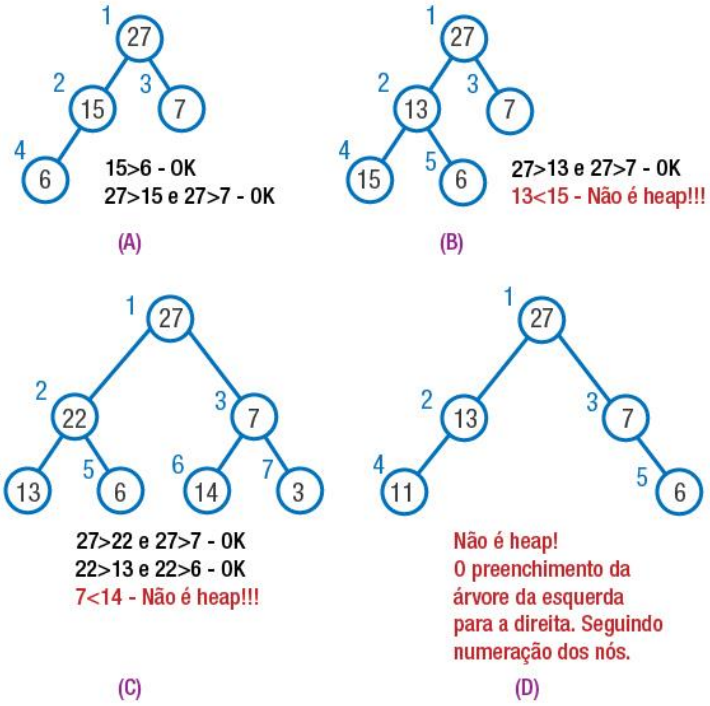


O heap ainda deve atender a algumas **regras estruturais**:

- um nó pode ter no máximo dois nós filhos que sempre devem ser preenchidos da esquerda para a direita, seguindo a numeração dos nós.
- o valor do nó pai deve ser sempre maior que o maior valor presente no nós filhos.

10

Se alguma das regras citadas não for respeitada, então a árvore binária não é um heap. Vejamos a seguir exemplos de aplicação dessas regras.



11

Fundamentalmente o método de ordenação **heapsorting** distribui os elementos do conjunto a ser ordenado em uma árvore seguindo as**regras estruturais de um heap**:

- preenchimento de cima para baixo e da esquerda para a direita, respeitando sempre o máximo de dois nós-filho.

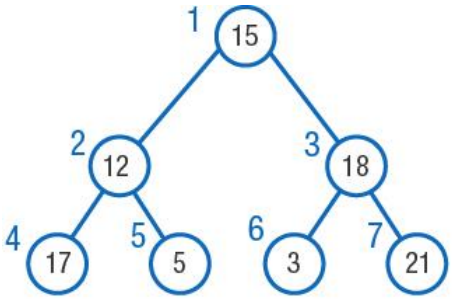
O método de ordenamento consiste em:

1. Criar um heap com os valores não ordenados para identificarmos o maior valor conjunto.
2. Extrair o valor do nó 1 e colocá-lo em uma lista ordenada.
3. repetir os passos 1 e 2 até que todos os elementos estejam na lista ordenada.

Para ilustrar esse método iremos ordenar o conjunto abaixo, utilizando a técnica de heap:

Valores = {15,12,18,17,5,3,21}

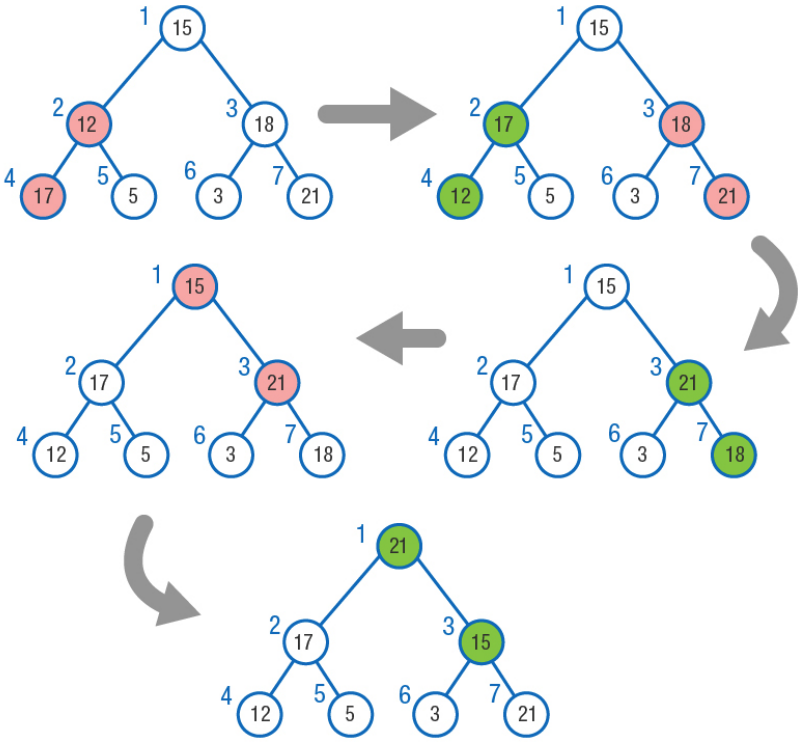
O primeiro passo é a montagem da árvore binária:



Observe que a regra estrutural I foi obedecida, mas não a regra II.

12

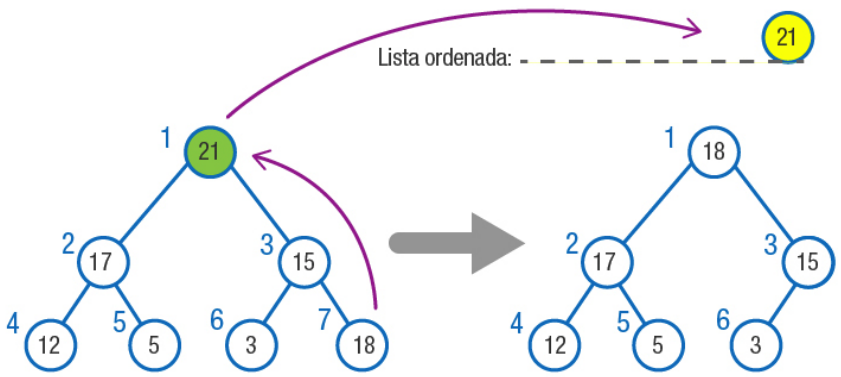
Como a regra estrutural I foi obedecida, mas não a regra II, passemos à troca de posições dos valores para que o nó-pai seja sempre maior que os valores dos nós-filho, começando de baixo para cima, conforme ilustração a seguir:



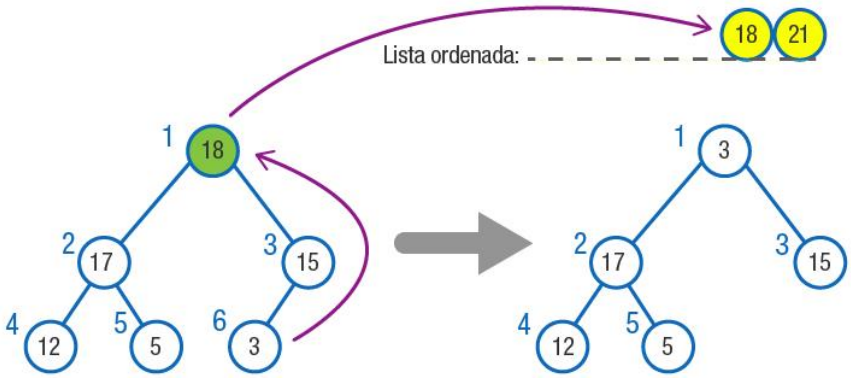
Podemos verificar que ao final da elaboração do primeiro heap, temos o maior valor na posição 1.

13

O maior valor (posição 1) será extraído do heap e colocado na última posição da lista ordenada. Em seu lugar colocaremos o último nó:

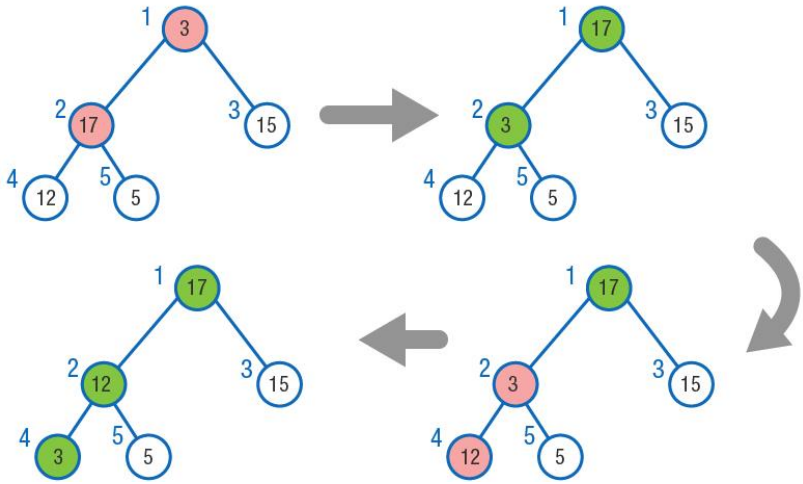


Podemos verificar que ao final dessa primeira iteração, temos uma nova árvore binária. O que faremos a seguir é repetir o procedimento anterior para obter o próximo elemento da lista ordenada. Por sinal, nesse caso, a árvore binária já um heap e, portanto, não será necessário realizar nenhuma movimentação, logo:

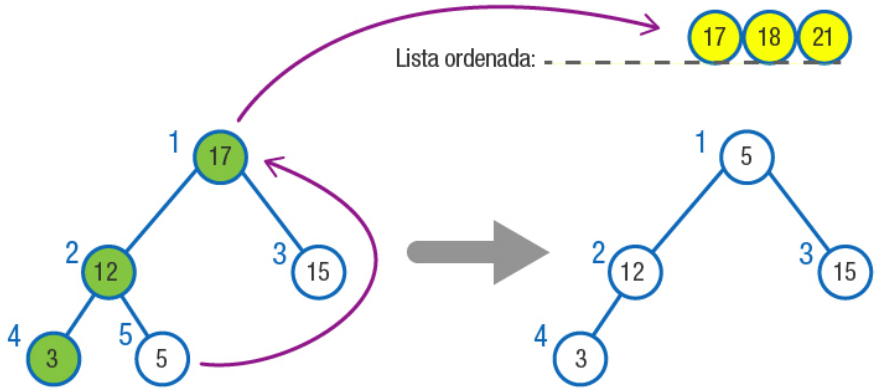


14

Seguimos novamente o procedimento, fazendo as trocas necessárias para transformar a árvore em heap:

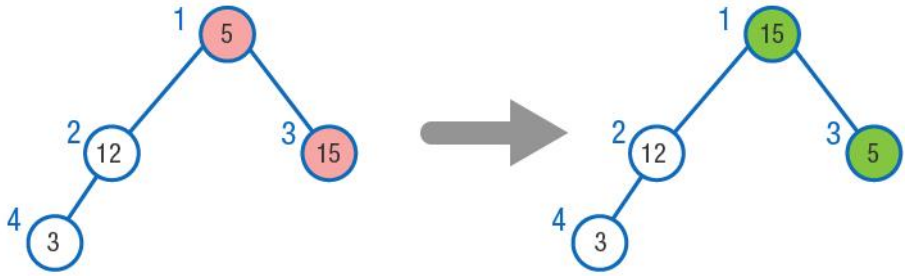


E mais uma vez passamos a extração do elemento do nó 1 para a lista ordenada:

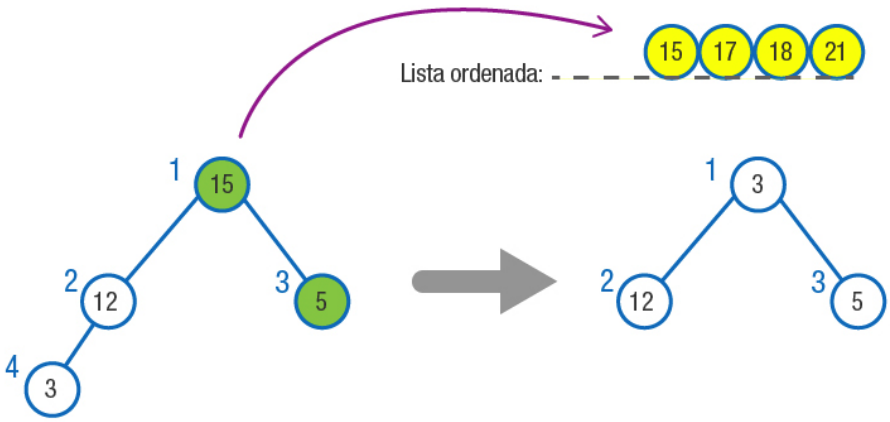


15

Seguimos em frente para a nossa quarta iteração, fazendo as trocas necessárias de forma a transformar a árvore binária em um heap.

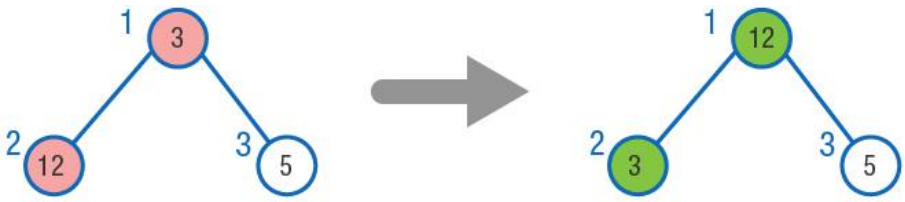


Novamente passamos à extração do maior elemento:

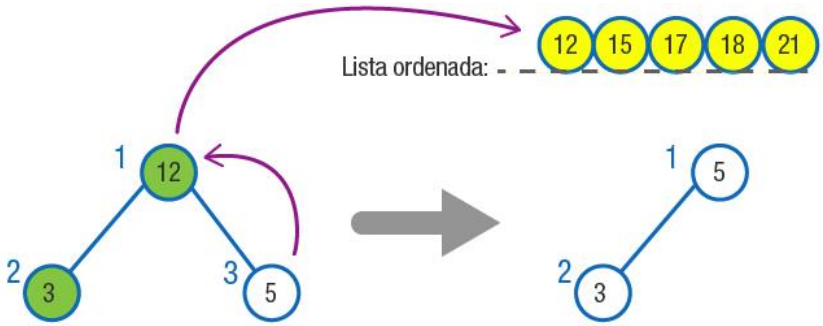


16

Passamos então para a próxima iteração:



E acrescentamos mais um valor na lista ordenada:



Logo em seguida, adicionamos os dois últimos números que já estão ordenados. Assim chegamos à lista ordenada:

3,5,12,15,17,18,21

17

Vamos ver agora como implementar esse método em Java. Para isso, vamos implementar uma classe chamada Heap. A classe possui dois métodos principais:

O primeiro método é o constructo, que constrói um heap com N elementos:

Exemplo 1_3_5: exemplo de busca heapSort.

```
private void construct() {
    int current, maxChildIndex;
    boolean done;
    for (int i = (heap.length - 2) / 2; i >= 0; i--) {
        current = i;
        done = false;
        while (!done) {// constroi o heap
            // com o nó no índice i
            if (2 * current + 1 > heap.length - 1) {
            // o nó atual não tem filho, então pára
            done = true;
            } else {
              // o nó atual tem pelo menos um nó filho
              // busca o índice do maior filho
```

```
                maxChildIndex=maxChild(current, heap.length - 1);
                if (heap[current] < heap[maxChildIndex]) {
               // o nó filho é maior, então troca e continua
                  swap(current, maxChildIndex);
                  current = maxChildIndex;
                } else { // se todos os nós tiverem a condição
                  // de valor satisfeita então para
                  done = true;
                  }
                      }
           }
    assert isValidHeap(heap, i, heap.length - 1) : "Error: Construction
phase is not working "
                    + "correctly";
    }
        testPrint(heap.length); // TEMP
}
```

18

O segundo método é o **extract**, que extrai o nó raiz e reconstrói o heap com N-1 elementos. O processo é repetido até que todos os nós sejam extraídos.

Exemplo 1_3_6: exemplo de busca heapSort.

```
private void extract() {
       int current, maxChildIndex;
    boolean done;
    for (int size = heap.length - 1; size >= 0; size--) {
        // remove o nó raiz
        sortedList[size] = heap[0];
        // move o último nó para o nó raiz
        heap[0] = heap[size];
        // reconstroi o heap com menos um elemento
        current = 0;
        done = false;
        while (!done) {
            if (2 * current + 1 > size) {
            // o nó atual não tem nó filho, então pára.
                    done = true;
            } else {
            // o nó atual tem pelo menos um nó filho
            // busca o índice do maior filho
                 maxChildIndex = maxChild(current, size);
                 if (heap[current] < heap[maxChildIndex]) {
                // se o filho for maior então troca e continua
                swap(current, maxChildIndex);
                current = maxChildIndex;
```

```
                } else { // se todos os nós tiverem a condição
                  // de valor satisfeita então para
                  done = true;
              }
            }
          }
          testPrint(size); // TEMP
      }
  }
```

19

Temos a seguir alguns métodos que são compartilhados entre os dois métodos principais:

- **maxChild**, que retorna o índice do filho de maior valor.
- **Swap**, que troca a posição de dois elementos.

O método que inicializa a lista é o **setData**, que copia a lista original em um atributo da classe. A seguir apresentamos o código completo incluindo a função Main com os dados de teste.

Exemplo 1_3_7: exemplo de busca heapSort.

```
public class Heap {
    private int[] heap;
    private int[] sortedList;
/**
 *  inicializa a lista de dados a serem ordenados
 * @param data lista de dados
 */
    public void setData(int[] data) {
        heap = new int[data.length];
        sortedList = new int[data.length];
        for (int i = 0; i < data.length; i++) {
            heap[i] = data[i];
        }
    }

    /**
   * ordena os elementos de uma lista
   *
   * @param não
   *            recebe parâmetros
   * @return retorna a lista ordenada
   */
    public int[] sort() {
        construct(); // implementa o Heap
        extract(); // Extrai o maior valor do Heap
        return sortedList;
```

```
    }

    /**
   * constroi o Heap
   *
   * @param não possui parâmetros
   * @return não possui valor de retorno
   */
    private void construct() {
        int current, maxChildIndex;
        boolean done;
        for (int i = (heap.length - 2) / 2; i >= 0; i--) {
            current = i;
            done = false;
            while (!done) {// constroi o heap
                // com o nó no índice i
                if (2 * current + 1 > heap.length - 1) {
                    // o nó atual não tem filho, então pára
                    done = true;
                } else {
                    // o nó atual tem pelo menos um nó filho
                    // busca o índice do maior filho
                    maxChildIndex = maxChild(current, heap.length - 1);
                    if (heap[current] < heap[maxChildIndex]) {
                        // o nó filho é maior, então troca e continua
                        swap(current, maxChildIndex);
                        current = maxChildIndex;
                    } else { // se todos os nós tiverem a condição
                        // de valor satisfeita então para
                        done = true;
                    }
                }
            }
            assert isValidHeap(heap, i, heap.length - 1) : "Error:
Construction phase is not working "
                    + "correctly";
        }
        testPrint(heap.length); // TEMP
    }

    /**
   * Extrai os valores do Heap
   *
   * @param não possui parâmetros
   * @return não possui valor de retorno
   */
    private void extract() {
        int current, maxChildIndex;
        boolean done;
```

```
        for (int size = heap.length - 1; size >= 0; size--) {
            // remove o nó raiz
            sortedList[size] = heap[0];
            // move o último nó para o nó raiz
            heap[0] = heap[size];
            // reconstroi o heap com menos um elemento
            current = 0;
            done = false;
            while (!done) {
                if (2 * current + 1 > size) {
                    // o nó atual não tem nó filho, então pára.
                    done = true;
                } else {
                    // o nó atual tem pelo menos um nó filho
                    // busca o índice do maior filho
                    maxChildIndex = maxChild(current, size);
                    if (heap[current] < heap[maxChildIndex]) {
                        // se o filho for maior então troca e continua
                        swap(current, maxChildIndex);
                        current = maxChildIndex;
                    } else { // se todos os nós tiverem a condição
                        // de valor satisfeita então para
                        done = true;
                    }
                }
            }
            testPrint(size); // TEMP
        }
    }
    /**
   * calcula o índice do maior nó filho
   * @param location índice do nó pai
   * @param end maior índice da lista
   * @return índice do maior nó filho
   */
    private int maxChild(int location, int end) {
        int result, leftChildIndex, rightChildIndex;
        rightChildIndex = 2 * location + 2;
        leftChildIndex = 2 * location + 1;

        assert leftChildIndex <= end : "Error: node at position " + location
                + "has no children.";
        if (rightChildIndex <= end
                && heap[leftChildIndex] < heap[rightChildIndex]) {
            result = rightChildIndex;
        } else {
            result = leftChildIndex;
        }
        return result;
```

```
    }
/**
 * Verificar se o heap obtido é válido
 * @param heap conjunto de valores do heap
 * @param start índice do primeiro nó
 * @param end índice do último nó.
 * @return
 */
    private boolean isValidHeap(int[] heap, int start, int end) {
        for (int i = start; i < end / 2; i++) {
            if (heap[i] < Math.max(heap[2 * i + 1], heap[2 * i + 2])) {
                return false;
            }
        }
        return true;
    }
/**
 * Troca a posição de dois elementos
 * @param loc1 índice do primeiro elemento
 * @param loc2 índice do segundo elemento
 */
    private void swap(int loc1, int loc2) {
        int temp;
        temp = heap[loc1];
        heap[loc1] = heap[loc2];
        heap[loc2] = temp;
    }
/**
 * Imprime todos os valores
 * @param limit maior índice
 */
    private void testPrint(int limit) {
        for (int i = 0; i < limit; i++) {
            System.out.print(" " + heap[i]);
        }
        System.out.println(" ");
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int[] number = { 90, 44, 84, 12, 77, 23, 38, 5, 17 };
        int[] sortedList;
        Heap heap = new Heap();
        heap.setData(number); // assign the original list
        sortedList = heap.sort();// sort the list
        for (int i = 0; i < sortedList.length; i++) { // print out
            System.out.print(" " + sortedList[i]); // the sorted
        }

    }
```

```
}
```

20

A implementação do método heapsort é mais complexa, porém é um método muito eficiente quando trabalhamos com uma quantidade muito grande de valores.

É importante salientar que apesar de termos feito uma implementação com números inteiros, nada impede de reescrever os métodos usando valores decimais. O método do heapsort melhora a performance dos métodos anteriores, visto que ele precisa de aproximadamente 1.5 N $\log_2$N para ordenar os elementos.

Apresento abaixo uma tabela com o número de comparações a serem realizadas para cada método:

N	Bubble Sort	Heap Sort
10	100	50
50	2.500	423
100	10.000	997
1000	1.000.000	14.949
10000	100.000.000	199.316
100000	10.000.000.000	2.491.446
1000000	1.000.000.000.000	29.897.353

Logo, podemos verificar que **mesmo para uma quantidade reduzida de valores, o método heapsort é muito mais eficiente**.

21

RESUMO

Nesse módulo apresentados os conceitos de pesquisa e ordenação. A pesquisa visa à recuperação de uma informação em uma lista. O método mais simples de pesquisa é a pesquisa linear, que verifica todos os elementos da lista para saber se correspondem ao valor procurado. Esse método se mostra ineficiente quando o valor procurado não se encontra na lista, visto que nesse caso teremos que comparar o valor com os N elementos da lista para obter uma resposta negativa da pesquisa. Um método mais eficiente é a pesquisa binária, que subdivide a lista em subconjuntos de valores cada vez menores até que o valor procurado seja encontrado. A desvantagem desse método é a necessidade da lista estar ordenada.

A necessidade de ordenar elementos de uma lista é muito comum em programação. O método mais simples consiste em ordenar os valores na lista buscando sempre o maior valor dos elementos não ordenados e colocando-o no final desse grupo. Esse é um método lento e custoso. O método chamado

de método bolha (*bubble sort*) é muito utilizado. Esse método consiste na ordenação dos elementos de dois em dois. É um método que não é mais eficiente em termos da quantidade de comparações, mas é computacionalmente mais simples de ser implementado.

O último método de ordenação que vimos foi o método heapsort. Esse método baseia-se numa estrutura complexa chamada de heap. O heap consiste em uma árvore binária com regras estruturais específicas:

1. valor do nó pai é sempre maior ou igual ao maior valor dos nós filhos.
2. o heap é sempre preenchido da esquerda para a direita.

O heapsort é o método mais eficiente de ordenação, mas a sua implementação computacional é um pouco mais complexa que a dos métodos anteriores.

UNIDADE 2 – ELEMENTOS BÁSICOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

MÓDULO 4 – ENTRADA E SAÍDA

01

1 - ARQUIVOS: OBJETOS FILE E JFILECHOOSER

Quem já não teve uma experiência infeliz quando estava trabalhando em um documento e por alguma razão o computador apagou e todo o trabalho foi perdido? Pois é, os dados armazenados em variáveis são perdidos quando o aplicativo é fechado e quando o computador é desligado. Sendo assim, a única forma de evitar isso é salvando o documento em um arquivo. O arquivo é uma forma de manter as informações de maneira permanente.

O objeto File é usado para acessar os arquivos tanto para escrita quanto para leitura. Para acessar um arquivo para a leitura precisaremos criar um objeto File. A sintaxe do método construtor de um novo objeto file é mostrado abaixo:

```
File <nome_do_objeto> = new File(<nome_de_arquivo>);
```

O arquivo deve estar no diretório da aplicação. Caso o arquivo que se deseje acessar não esteja nesse diretório padrão então o arquivo deverá ser acompanhado do caminho completo do arquivo:

```
File <nome_do_objeto> = new File(<caminho_do_arquivo>,<nome_de_arquivo>);
```

Para garantir a compatibilidade com os diversos sistemas operacionais, o caminho deve ser sempre informado utilizando o caractere /. Vejamos o exemplo abaixo:

```
File meuArquivo = new File("c:/program files/meuprograma","dados.txt");
```

02

O objeto File pode ser associado com um arquivo ou com o um diretório. É importante verificar se o objeto File foi associado corretamente. Para isso podemos utilizar o método isFile(), como segue:

```
if (inFile.exists( )) {
// se o objeto foi associado corretamente
} else {
// se o arquivo não foi associado corretamente.
}
```

No caso da associação com um diretório, poderemos obter o conteúdo do diretório como segue:

```
File meuDiretorio = new File(“C:/Program files”);
String conteudoDiretorio[] = Diretorio.list();
```

Nesse caso vem então outra necessidade. Como saber se um objeto File está associado a um arquivo ou a um diretório? O método da isFile() responde essa questão, conforme veremos no exemplo a seguir.

03

Exemplo _2_4_1: uso do objeto JFileChooser

```
File file = new File("C:/Program Files");
        if (file.isFile()) {
                    System.out.println("Arquivo!");
        } else {
                   System.out.println("Diretório!");
        }
```

Convenhamos, para o usuário é complicado escrever o caminho completo de um arquivo no programa. O objeto JFileChooser fornece ao usuário uma forma mais simples de selecionar arquivos ou diretórios. O JFileChooser faz parte do pacote javax.swing. O seguinte código permite mostrar uma caixa de diálogo:

```
JFileChooser chooser = new JFileChooser( );
    ...
    chooser.showOpenDialog(null);
```

Se não for colocado nenhum argumento na chamada do construtor do objeto então a caixa de diálogo abrirá no diretório “Meus Documentos”. Caso se queira que a caixa de diálogo abra em um diretório diferente, podemos usar o método setCurrentDirectory():

```
File file = new File("C:/Program Files");
    chooser.setCurrentDirectory(startDir);
```

```
    ...
    chooser.showOpenDialog(null);
```

Apresentaremos a seguir um exemplo com os principais métodos do objeto JFileChooser.

04

Exemplo _2_4_2: uso do objeto JFileChooser

```
import java.io.*;
import javax.swing.*;

public class jfchooser {

     public static void main(String[] args) {
       JFileChooser chooser;
    File file, directory;
    int status;
    //Declaração do objeto JFileChooser
    chooser = new JFileChooser();
    status = chooser.showOpenDialog(null);
    //Abre a caixa de diálogo e mostra os dados do arquivo selecionado
    if (status == JFileChooser.APPROVE_OPTION) {
        //busca o arquivo selecionado
        file = chooser.getSelectedFile();
        //busca o diretório selecionado
        directory = chooser.getCurrentDirectory();
        //Imprime as informações
        System.out.println("Directory: " + directory.getName());
        System.out.println("File selected to open: " + file.getName());
        //imprime o caminho completo do arquivo selecionado
        System.out.println("Full path name: " + file.getAbsolutePath());
    } else {
        System.out.println("Open File dialog canceled");
    }
    System.out.println("\n\n");
    //Abre a caixa de diálogo com o botão "Salvar"
    status = chooser.showSaveDialog(null);
    if (status == JFileChooser.APPROVE_OPTION) {
        file = chooser.getSelectedFile();
        directory = chooser.getCurrentDirectory();
        System.out.println("Directory: " + directory.getName());
        System.out.println("File selected for saving data: "+
file.getName());
        System.out.println("Full path name: " + file.getAbsolutePath());
    } else {
        System.out.println("Save File dialog canceled");
    }
   }
```

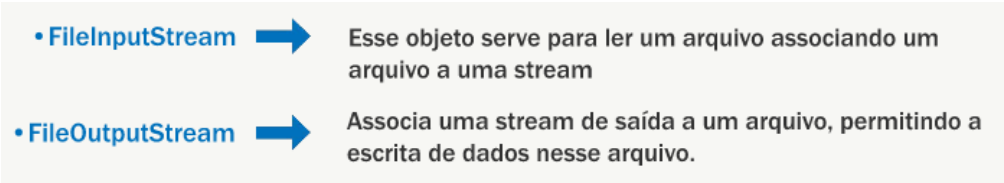
```
}
```

05

2 - FLUXOS (STREAMS)

Chamamos de stream uma sequência de dados de informação. As streams podem ser de dois tipos stream de entrada e stream de saída. A stream de entrada possui uma origem que geralmente é um arquivo. A stream de saída possui um destino que irá receber os dados da stream. Para que consigamos ler as informações contidas em um arquivo deveremos então associar uma stream a esse arquivo (stream de entrada). De forma análoga, para que consigamos escrever dados em um arquivo deveremos associar uma stream de saída a esse arquivo.

O java possui dois objetos básicos para associarmos streams a arquivos:



O primeiro passo é associar um objeto do tipo File a um arquivo, como fizemos na seção passada:

```
File outFile = new File("sample1.data");
```

O Segundo passo é associar esse arquivo a uma stream de saída:

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

O exemplo a seguir mostra como gravar um array de valores em um arquivo.

06

Exemplo_ 2_4_3: uso de streams

```
import java.io.*;
import javax.swing.*;

public class outputStream {

    public static void main(String[] args) throws IOException {
        JFileChooser chooser;
        File outfile = null;
        File infile = null;
        byte[] outbyteArray = {10, 20, 30, 40, 50, 60, 70, 80};
```

```
        chooser = new JFileChooser();
        int status = chooser.showSaveDialog(null);
        if (status == JFileChooser.APPROVE_OPTION) {
            outfile = chooser.getSelectedFile();
        } else {
            System.out.println("Save File dialog canceled");
        }
        FileOutputStream outStream = new FileOutputStream(outfile);
        outStream.write(outbyteArray);
        outStream.write(outbyteArray[0]);
        outStream.write(outbyteArray[4]);
        outStream.close();
        //Abre novamente o diálogo para abrir o arquivo
        status = chooser.showOpenDialog(null);
        infile = chooser.getSelectedFile();
        FileInputStream inStream = new FileInputStream(infile);
        //cria o array para receber os dados da stream
        int fileSize = (int) infile.length();
        byte[] inbyteArray = new byte[fileSize];
        //Mostra o array de valores do arquivo na tela
        inStream.read(inbyteArray);
        for (int i = 0; i < fileSize; i++) {
        System.out.println(inbyteArray[i]);
        }
    }
}
```

07

3 - Elementos avançados de I/O

- **Manipulando arquivos de texto**

Os objetos DataOutputStream e DataInputStream permitem, respectivamente, a gravação e a leitura de arquivos binários. O objeto DataOutputStream se encarrega de converter os tipos de dados primitivos do java em uma sequência de bytes e usa o objeto FIleOutputStream para escrever essas informações binárias no arquivo. De forma análoga, o objeto DataInputStream recebe do objeto FileInputStream uma sequência de bytes e se encarrega de convertê-la em informações no formato dos tipos primitivos de dados do java.

Também são muito utilizados os métodos *PrintWriter*, que recebe uma sequência de bytes e grava em formato de texto. O objeto*BufferedReader* permite a leitura de um arquivo de texto. Apresentamos a seguir um exemplo de gravação e leitura de um arquivo.

Exemplo_ 2_4_4: gravação e leitura de arquivos binários

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

```
public class exemplo_2_4_3 {
    public static void main (String[] args) throws IOException {
            //GRAVAÇÃO DO ARQUIVO
                //Configuração do arquivo e do objeto FileOutputStream
        File outFile = new File("teste.dat");
        FileOutputStream outFileStream = new FileOutputStream(outFile);
        PrintWriter outStream = new PrintWriter(outFileStream);
        //Escreve os dados para a sequencia de bytes
        outStream.println(1000);
        outStream.println('A');
        outStream.println(true);
        //Fecha a stream
        outStream.close();

                //LEITURA DO ARQUIVO
        File inFile = new File("teste.dat");
                //Configura o objeto BufferedReader
        FileReader fileReader = new FileReader(inFile);
        BufferedReader bufReader= new BufferedReader(fileReader);
                //Busca uma linha do arquivo de texto  e grava na variável
str
        String str = bufReader.readLine( );
                //converte o valor lido para o format original e imprime na
tela
        int i = Integer.parseInt(str);
        System.out.print(i);
        str = bufReader.readLine( );
        char c = str.charAt(0);
        System.out.print(c);
        bufReader.close();
       }
}
```

08

- **Objeto Scanner**

O objeto scanner foi muito utilizado por nós na unidade I para a leitura de dados do teclado, entretanto esse objeto pode ser associado a um arquivo. Nesse caso não será necessária a criação de uma stream, pois o objeto scanner já se encarregará da leitura e conversão das informações lidas do arquivo.

Exemplo_ 2_4_5: leitura do arquivo usando o objeto scanner.

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

public class exemplo_2_4_5 {
```

```
    public static void main (String args[]) throws FileNotFoundException,
    IOException {
    //Cria o objeto scanner e associa a um arquivo
    Scanner scanner = new Scanner(new File("teste.dat"));
    //busca valor inteiro
    int i = scanner.nextInt();
    //busca valor inteiro
    long l = scanner.nextLong();
    //busca float
    float f = scanner.nextFloat();
    //busca double
    double d = scanner.nextDouble();
    //busca char
    char c = scanner.next().charAt(0);
    //busca valor booleano
    boolean b = scanner.nextBoolean();
    System.out.println(i);
    System.out.println(l);
    System.out.println(f);
    System.out.println(d);
    System.out.println(c);
    System.out.println(b);
    //fecha o objeto scanner
    scanner.close();
    }
}
```

09

- **Lendo e escrevendo objetos para um arquivo**

A linguagem java permite a leitura e escrita direta de objetos em arquivos. O objeto deve, necessariamente, implementar a interface*Serializable*. Para isso basta incluir na declaração da classe a implementação conforme exemplo abaixo:

```
classe <nome_da_classe>  implements Serializable {
}
```

Adicionando essa implementação o objeto poderá ser usado em operações de entrada e saída de arquivos. Para manipular a leitura e escrita de objetos em streams utilizamos os objetos inObjectStream e OutObjectStream, respectivamente. Apresentamos abaixo um exemplo que ilustra a gravação de objetos seguida da leitura e impressão na tela.

Exemplo_ 2_4_6: leitura do arquivo usando o objeto scanner.

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

```
public class exemplo_2_4_6 {

    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        // Criação de um array de objetos do tipo pessoa
        pessoa[] pessoas = new pessoa[5];
        pessoas[0]= new pessoa("André","Sr.",'M');
        pessoas[1]= new pessoa("Paulo","Sr.",'M');
        pessoas[2]= new pessoa("Maria","Sra.",'F');
        pessoas[3]= new pessoa("Aline","Srs.",'F');
        pessoas[4]= new pessoa("Carlos","Dr.",'M');

        //GRAVAÇÃO DOS OBJETOS EM UM ARQUIVO
        File outFile= new File("objects.dat");
        FileOutputStream outFileStream= new FileOutputStream(outFile);
        ObjectOutputStream outObjectStream= new
ObjectOutputStream(outFileStream);
        for(int i=0;i<5;i++){
            outObjectStream.writeObject(pessoas[i]);
        }
        outObjectStream.close();

        //LEITURA DAS INFORMAÇÔES NO ARQUIVO

        pessoa[] pessoaslidas = new pessoa[5];
        File inFile= new File("objects.dat");
        FileInputStream inFileStream= new FileInputStream(inFile);
        ObjectInputStream inInputStream= new
ObjectInputStream(inFileStream);
        for (int i=0;i<5;i++){
                        //Lê cada objeto e coloca no array de entrada.
            pessoas[i] = (pessoa) inInputStream.readObject();
            System.out.println(pessoas[i].Nome);
            System.out.println(pessoas[i].Tratamento);
            System.out.println(pessoas[i].sexo);
        }
        inInputStream.close();
    }
}



public class pessoa  implements Serializable{

    public String Nome;
    public String Tratamento;
    public char sexo;

    public pessoa (String Nome, String Tratamento, char sexo){
```

```
        this.Nome=Nome;
        this.Tratamento = Tratamento;
        this.sexo = sexo;
    }
}
```

10

RESUMO

As informações que estão armazenadas na memória em forma de variáveis são perdidas quando a aplicação é fechada ou mesmo quando o computador é desligado. Sendo assim, é imprescindível saber como gravar essas informações de arquivos.

A linguagem java possui o objeto File que consiste em uma representação abstrata de um arquivo independente da interface do usuário e do sistema operacional. Vimos que esse objeto possui diversos métodos que permitem verificar se determinado arquivo existe, se é realmente um arquivo ou um diretório, bem como diversas outras funcionalidades. Para acessarmos o conteúdo do arquivo deveremos associar essa representação abstrata a um objeto do tipo stream que será responsável por receber as informações do arquivo e transformá-las em um array de bytes.

Vimos nesse módulo os seguintes objetos do tipo stream:

- FileInputStream e FileOutputStream – servem, respectivamente, para a escrita e leitura de arquivos binários.
- DataOutputStream e DataInputStream – servem, respectivamente, para a escrita e leitura de arquivos de texto.
- inObjectStream e OutObjectStream – servem, respectivamente, para a leitura e escrita de objetos.

Além desses objetos, vimos ainda que o objeto scanner, utilizado para ler informações do teclado, também pode ser associado a um arquivo físico permitindo a sua leitura formatada.