Programação em Java

Djonathan Luiz de Oliveira Quadras

14/04/2020

Contents

Bem vindo! **7**

Sobre Programação **9**

Sobre o Livro 9

Ferramentas Computacionais 9

1 Lógicas de Programação **11**

1.1 Pseudocódigos 11

2 Princípios da Programação em Java **13**

2.1 Classes 14

2.2 Objetos 14

2.3 Variáveis 14

2.4 Criando o primeiro código em Java 15

3 Operadores **17**

3.1 Operações Matemáticas 17

3.2 Operadores Relacionais 18

3.3 Lógica Booleana e Operadores Lógicos 18

4 Extruturas de Decisão ou Seleção **21**

4.1 If-Else 22

4.2 Operador Ternário 22

4.3 Switch 23

Exercícios 24

5 Estruturas de Repetição **27**

5.1 For 27

5.2 While 27

5.3 Do-While 28

Exercicios 28

6 Classes e Interfaces Importantes em Java **31**

6.1 Classes 31

6.2 Interfaces 32

7 Vetores, Matrizes e Listas **35**

7.1 Arrays 36

7.2 Matrizes 40

7.3 Listas 41

Exercícios 44

8 Coleções **45**

8.1 Maps 45

8.2 Sets e HashSets 45

8.3 StockList 45

9 Tópicos Avançados em Java **47**

9.1 Classes, Construtores e Herança 47

9.2 Composição, Encapsulamento e Polimorfismo 47

9.3 Classes Abstratas e Interfaces 47

10 Pesquisa Operacional e Solução de Problemas **49**

10.1 Heurísticas e Meta-Heurísticas 49

11 Aplicações em Logística e Cadeia de Suprimentos **51**

11.1 Cadeia de Suprimentos Varejista 51

11.2 Integração de Fornecedores, Produção e Distribuição 51

12 Problemas Clássicos de Computação **53**

Considerações Finais **55**

Respostas dos Exercícios **57**

Bem vindo!

Bem vindo ao livro de programação em Java!

Como você vai perceber, o livro ainda está em desenvolvimento e pode ter diversos erros (além de estar incompleto).

Para casos de dúvidas ou sugestões sinta-se a vontade para entrar em contato comigo!

email: oliveira.ind.eng@gmail.com

Enjoy your visit! :)

Figure 1: I'm working on it!

Sobre Programação

Working on it! :)

Sobre o Livro

Working on it! :)

Ferramentas Computacionais

Working on it! :)

Chapter 1

Lógicas de Programação

Working on it! :)

1.1 Pseudocódigos

Working on it! :)

Chapter 2

Princípios da Programação em Java

A linguagem Java é considerada como sendo de baixo nível, ou seja, sua programação (e consequentemente sua leitura) não é trivial como se estivéssemos escrevendo um livro. Ela se apresenta como sendo uma linguagem muito burocrática e que necessita de diversos processos para que funcione corretamente. Um código em Java sempre deverá ser montado com uma estrutura semelhante a apresentada na figura abaixo.

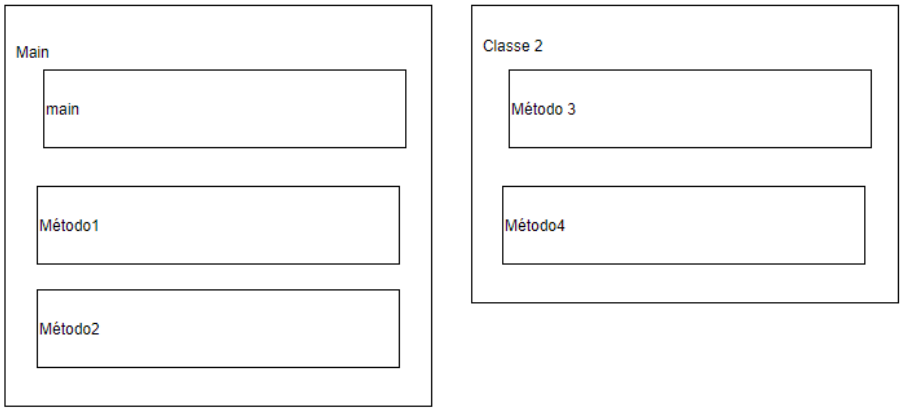

Figure 2.1: Classes

Um código é geralmente composto por Classes e Objetos. Esses pontos serão explicados nas subseções a seguir.

2.1 Classes

As classes de programação são receitas de um objeto, aonde têm características e comportamentos, permitindo assim armazenar propriedades e métodos dentro dela. Uma classe geralmente representa um substantivo, por exemplo uma pessoa, um lugar ou um sistema.

Vamos imaginar uma universidade. Ela é composta por diversas classes: alunos, professores, prédios, salas de aula. Note que uma classe pode ser parte de uma outra classe, ou seja, todas as classes apresentadas fazem parte de uma classe maior chamada de universidade. Mas afinal, o que diferencia uma classe da outra? A diferenciação, além do nome dado, está nos objetos que compõe a classe.

2.2 Objetos

Objetos são caracterizados por atributos e métodos e são características definidas pelas classes.

2.2.1 Atributos ou Propriedades

Atributos são as características de um objeto. Essas características também são conhecidas como variáveis. Utilizando o exemplo dos cães, temos alguns atributos, tais como: cor, peso, altura e nome.

2.2.2 Métodos

Métodos são as ações que os objetos podem exercer quando solicitados, onde podem interagir e se comunicarem com outros objetos. Utilizando o exemplo dos cães, temos alguns exemplos: latir, correr, pular.

2.3 Variáveis

Variáveis são uma forma de salvar informações no computador. As variáveis que nós armazenamos em um programa são acessadas pelo nome que nós atribuímos a elas.

Uma variável pode ter seu valor alterado ao longo do programa, ou seja, seu valor é variável. É apenas necessário informar ao computador qual o tipo de informação será armazenado e, então, dar um nome ao valor. Existem muitos tipos de dados que podem ser utilizados para definir uma variável. Os principais estão listados abaixo.

Table 2.1: Tipos de Variáveis

Tipo	Descrição
int	Números Inteiros
double	Números Decimais
boolean	Armazena apenas 'Verdadeiro' ou 'Falso'
String	Frases
char	Caracteres

Existem diversos outros tipos de variáveis, que serão apresentados com maior detalhe nas seções seguintes.

2.4 Criando o primeiro código em Java

A primeira etapa para a criação de um código em Java é a criação de uma classe. Por padrão, a classe principal do seu programa se chama "Main".

```
public class Main{
  //Oi! Eu sou uma classe!
}
```

Perceba alguns pontos importantes:

- O código inicia com a palavra "*public*". Isso indica que sua classe pode ser encontrada por outras classes que estiverem salvas no mesmo diretório. Você pode alterar por "*private*" ou "*protected*";
- Há a presença de chaves `{ }` no código. Isso é uma burocracia da linguagem. A classe é correspondente ao que está dentro das chaves. Tudo o que estiver fora delas não será considerado como parte do código;
- Para fazer comentários em Java, usa-se duas barras `//` .

Agora que temos a nossa classe criada, necessitamos criar os objetos. Por agora vamos criar apenas um método.

```
public class Main{
  public static void main(String[] args){
    System.out.println("Hello World!");
  }
}
```

Agora, dentro da nossa classe foi criado o método main. Isso também segue um padrão imposto pela linguagem. Há, novamente, alguns pontos que merecem destaque.

- A função "System.out.println()" é utilizada para escrever algum texto no console;
- Todas as linhas de código devem ser terminadas com um ponto e vírgula.

Colocando este código para correr, deverá aparecer no console a mensagem "*Hey, Hello World*!". Pronto! Você acaba de criar seu primeiro código!

2.4.1 Criando Variáveis

Working on it! :)

Chapter 3

Operadores

Um artifício essencial para quem programa é a utilização de operadores. Sejam eles matemáticos, lógicos ou relacionais, é praticamente impossível desenvolver um código sem utilizar um operador. Nessa seção serão apresentados e explicados os operadores em linguagem Java.

3.1 Operações Matemáticas

É muito comum utilizarmos as operações matemáticas em programação. Frequentemente estaremos somando ou subtraindo duas variáveis. Com isso, torna-se necessário conhecer os operadores matemáticos em java. Os básicos estão listados na tabela abaixo.

Existem também os operadores incrementais. Estes operadores são utilizados quando queremos acrescentar (ou diminuir) uma unidade no valor de uma variável. Eles estão listados na tabela abaixo.

Table 3.1: Operações Matemáticas

Operador	Uso	Descrição
+	x + y	Soma x e y
-	x - y	Subtrai y de x
*	x*y	Multiplica x por y
/	x/y	Divide x por y
%	x%y	Calcula o resto da divisão de x por y

Table 3.2: Operações Matemáticas Úteis

Operador	Uso	Descrição
++	b++	Incrementa 1 em b, avaliando b antes de incrementar
++	b++	Incrementa 1 em b, avaliando b depois de incrementar
–	b–	Decrementa 1 em b, avaliando b antes de incrementar
–	b–	Decrementa 1 em b, avaliando b depois de incrementar

Table 3.3: Operações Relacionais

Operador	Uso	Descrição
>	x > y	x é maior que y
>=	x >= y	x é maior ou igual a y
<	x < y	x é menor que y
<=	x <= y	x é menor ou igual a y
==	x == y	x é igual a y
!=	x != y	x é diferente de y

3.2 Operadores Relacionais

Os operadores relacionais são utilizados quando queremos estabelecer relações entre as variáveis, ou seja, quando queremos comparar uma variável com a outra. Eles estão apresentados na tabela abaixo.

Para utilizar outras ferramentas matemáticas, é necessário invocar a classe matemática do Java. Para isso, faz-se `double valor = Math.` e seleciona-se a função desejada (irá aparecer automaticamente uma lista com todas as opções possíveis. Por exemplo, é possível calcular a raiz quadrada, exponencial e cosseno conforme apresentado a seguir.

```
double raiz = Math.sqrt(9); // raiz quadrada
double exponencial = Math.exp(2); // e elevado ao quadrado
double cosseno  = Math.cos(Math.PI); //cosseno de pi em radianos
```

3.3 Lógica Booleana e Operadores Lógicos

A lógica booleana foi desenvolvida no século 19 pelo matemático George Boole como um esquema para usar métodos algébricos na formalização da lógica e raciocínio. A álgebra booleana trabalha com dois números, 1 e 0. Para a programação, trataremos 1 como sendo “Verdadeiro” e 0 como sendo “falso”.

As funções relacionais sempre gerarão resultados booleanos. Dessa forma, é possível utilizar operadores lógicos para trabalhar com as informações e gerar novos

Table 3.4: Operações Matemáticas Úteis

Operador	Descrição
&	AND lógico
| ou ||	OR lógico
^	XOR lógico
!	NOT lógico

Table 3.5: Operações com Operador AND

Valor 1	Operador	Valor 2	Resultado
true	&	true	true
true	&	false	false
false	&	false	false

resultados. Os operadores lógicos são amplamente utilizados quando queremos criar expressões lógicas. Os operadores lógicos estão apresentados na tabela abaixo.

3.3.1 Operador AND

O operador AND, ou "e", retorna "verdadeiro" só e somente só quando está comparando dois booleanos com valor "verdadeiro.

3.3.2 Operador OR

O operador OR, ou "ou", retorna "verdadeiro" quando está comparando dois booleanos em que ao menos um tenha valor "verdadeiro.

3.3.3 Operador XOR

O operador XOR, ou "ou exclusivo", retorna "verdadeiro" se só e somente só estiver comparando dois booleanos em que somente um dos valores seja "ver-

Table 3.6: Operações com Operador OR

Valor 1	Operador	Valor 2	Resultado
true	|	true	true
true	|	false	true
false	|	false	false

Table 3.7: Operações com Operador XOR

Valor 1	Operador	Valor 2	Resultado
true	^	true	false
true	^	false	true
false	^	false	false

Table 3.8: Operações com Operador XOR

Operador	Valor	Resultado
!	true	false
!	false	true
!!	true	true

dadeiro.

3.3.4 Operador NOT

O operador NOT, ou "não", retorna o valor contrário ao valor escolhido. Note que este operador compara dois valores, apenas muda de verdadeiro para falso e vice-versa.

##Exercícios {-}

Chapter 4

Extruturas de Decisão ou Seleção

É comum nos códigos encontrarmos situações como a situação abaixo.

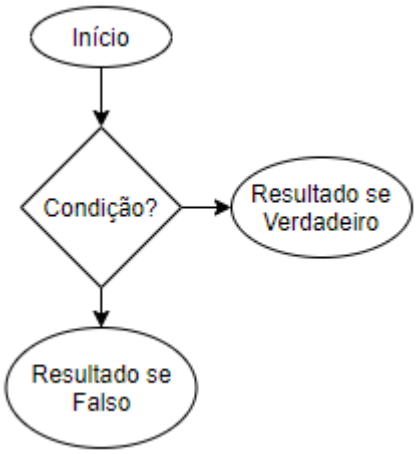

Figure 4.1: Decisores

Ou seja, o programa irá tomar uma decisão diferente para condições diferentes. Existem duas formas de selecionar em Java, que serão explicadas nas subseções seguintes.

4.1 If-Else

A seleção por if tem a forma apresentada abaixo. Depois da palavra-chave if é necessária uma expressão booleana entre parênteses. Caso a expressão booleana resulte no valor true em tempo de execução então o bloco seguinte será executado, caso resulte em false aquele será ignorado.

```
if(condiçãoBooleana){
  //Instrução a ser executada caso a condiçãoBooleana seja verdadeira.
}
```

Existem também as variações do método if. Uma delas é o if/else. A instrução de seleção dupla if/else tem função complementar à de if: executa instruções no caso da expressão booleana de if resultar em false.

```
if (condiçãoBooleana){
  //instruções que serão executadas caso a condiçãoBooleana resulte true.
} else {
  //instruções que serão executadas caso a condiçãoBooleana resulte false.
}
```

Complementar ao "if/else" temos o operador "else if". Esse recurso possibilita adicionar uma nova condição à estrutura de decisão para atender a lógica sendo implementada.

```
if (condiçãoBooleana) {
    //instruções que serão executadas caso a condiçãoBooleana resulte true.
} else if(outraCondiçãoBooleana){
    //instruções que serão executadas caso a outraCondiçãoBooleana resulte true.
}
```

4.2 Operador Ternário

O operador ternário é uma forma compacta de se montar um operador *if-else*. Ele é utilizado para alocar valor em alguma determinada variável. Diferentemente do *if-else*, ele **não** realiza operações. Sua estrutura é igual ao apresentado abaixo.

```
var = (lógica booleana) ? (Valor 1):(Valor 2)
```

Ou seja, a variável `var` receberá qualquer um dos dois valores dependendo do resultado da lógica booleana, recebendo o `Valor 1` em caso de resultado igual a `true` e `Valor 2` em caso de resultado igual a false. Perceba o caso abaixo.

```
public class Main {
    public static void main(String[] args) {
        int a = 7;
        int b = 6;
        String valor;
        valor = a > b ? "a é maior que b":"a é menor que b";
    }
}
```

Neste caso, como `a` de fato é maior que `b`, a lógica booleana resultará em um valor igual a `true`, retornando para a variável valor a string `"a é maior que b"`.

4.3 Switch

O operador *switch* é muito útil para a tomada de decisões em um programa. Sua estrutura se assemelha à uma estrada, onde os caminhos já estão pré-determinados. Dessa forma, a decisão de qual caminho seguir é baseada no valor da entrada do decisor.

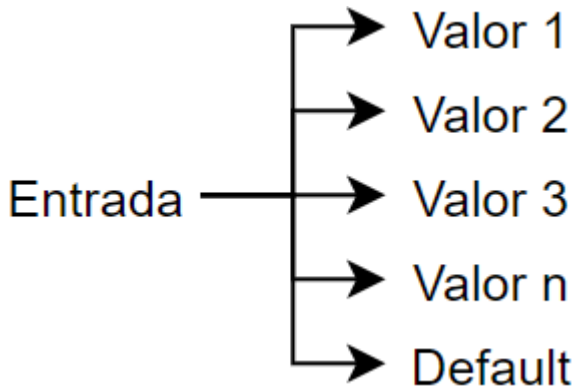

Figure 4.2: Switch

A estrutura do sistema de decisão igual ao apresentado abaixo. A função `switch` irá avaliar a variável `var`. Cada `case` corresponde a um possível valor que `var` pode assumir. Caso o valor de `var` não esteja presente em nenhuma das opções, o `switch` irá para a opção `default` que é a resposta padrão.

```
switch (var){
    case Valor1:
        //Realiza as essas operações
```

```
        break;
    case Valor2:
        //Realiza as essas operações
        break;
    default:
        //Realiza as essas operações paadrão
        break;
}
```

A variável `var` pode ser de qualquer tipo (*int, double, string...*). Perceba o código apresentado abaixo. Se correr o código, será apresentado no console a resposta `"Muito!"`.

```
public class Main {
    public static void main(String[] args) {
        String var = "Disposto?";
        switch (var){
            case "Oi, tudo bem?":
                System.out.println("Tudo! E com voce?");
                break;
            case "Bom dia!":
                System.out.println("Flor do Dia!");
                break;
            case "Disposto?":
                System.out.println("Muito!");
        }
    }
}
```

Exercícios

1. Qual a diferença entre uma estrutura de decisão estilo *if-else* e um Operador Ternário?
2. Crie um código em Java que contenha uma variável do tipo inteiro e que imprima no console se o número é par ou ímpar.
3. Faça um programa que contenha três variáveis: dinheiro, preço e limite do cartão. A variável "dinheiro" representa a quantidade de dinheiro presente na carteira, "preço" é o preço de um lanche e "limite no cartão" é o limite disponível para compras no cartão de crédito. O processo de análise é feito da seguinte forma (note que deve-se retornar, em caso de compra, qual o a opção de pagamento escolhida):

- Se houver dinheiro suficiente disponível na carteira, a compra será feita totalmente em dinheiro;

- Caso contrário, se houver limite disponível, a compra deverá ser feita totalmente no cartão de crédito;
- Caso contrário, deve-se analisar se é possível comprar o lanche juntando os valores de dinheiro disponível e limite de cartão;
- Caso seja possível, deve-se pagar a totalidade do valor disponível em dinheiro e pagar o restante em cartão. Neste caso, deve-se dizer quanto será pago no cartão;
- Caso contrário, deve-se retornar a mensagem "Você não pode pagar, que pena!".

4. Utilizando a mesma situação do exercício anterior, acrescente agora a variável booleana "vontade", que é a vontade da pessoa em pagar o lanche utilizando o cartão. As análises que envolvem pagar com o cartão ficarão, então, em função dessa nova variável.
5. Faça um código que receba uma variável do tipo inteiro de valor entre 1 e 5, cada um correspondente a um dia útil da semana. O código deve imprimir no console o dia da semana correspondente. Caso a variável tenha um valor diferente, o programa deve imprimir no console a frase "Esse não é dia de feira!".
6. Faça um código semelhante ao anterior. Crie uma variável do tipo *string* chamada idioma e faça com que o programa responda no console nos seguintes idiomas: português, inglês e espanhol. Caso seja inserido outro idioma, o programa deve responder "Idioma não reconhecido".

Chapter 5

Estruturas de Repetição

Nas linguagens de programação as estruturas de repetição são utilizadas para executar um conjunto de instruções ou funções repetidamente baseando-se em uma condição que determinará a continuação ou não da repetição. Existem três tipos: *for*, *while* e *do-while*, que serão explicados abaixo.

5.1 For

A estrutura *for* é um comando de controle de fluxo que repete uma parte do código múltiplas vezes. É utilizado quando ocorrerá um número pré-determinado de repetições. Sua estrutura em Java é apresentada abaixo.

```
for(int i = valorMinimo; i <= valorMaximo; i++){
    //Código
}
```

Dentro do parêntesis há quatro informações que devem ser declaradas: qual a variável que será utilizada como controle da repetição, seu valor inicial, seu valor final (critério de parada) e, por fim, quanto deve ser incrementado ou decrementado a cada repetição à variável.

5.2 While

A estrutura `while` é um comando de controle de fluxo que executa uma parte do código múltiplas vezes baseado em uma dada condição booleana. É utilizado quando o número de repetições não é pré-determinado. A estrutura do código é apresentada abaixo.

```
while (erro > erroMaximo){
    //Código
}
```

Dentro do parêntesis há apenas uma informação que deve ser preenchida: a condição de parada. Essa condição é estabelecida por uma lógica booleana, geralmente comparando dois valores. No exemplo temos a comparação de dois erros pois esta é uma lógica clássica aplicada a este tipo de repetição.

5.3 Do-While

A estrutura `do-while` é um comando de controle e fluxo que executa uma parte do código pelo menos uma vez e a execução posterior depende da condição booleana especificada. É utilizado quando o número de iterações não é pré-determinado mas o código deve ser executado ao menos uma vez.

Note que a diferença de um código para o outro está simplesmente no fato de a lógica booleana na estrutura `while` ser avaliada antes de correr o código, enquanto na estrutura `do-while` ela é avaliada após correr uma vez. A estrutura do código é apresentada abaixo.

```
do{
    //Código
}while (erro > erroMaximo);
```

Dentro do parêntesis há apenas uma informação que deve ser preenchida: a condição de parada. Essa condição é estabelecida por uma lógica booleana, geralmente comparando dois valores. No exemplo temos a comparação de dois erros pois esta é uma lógica clássica aplicada a este tipo de repetição.

Exercicios

1. Qual a diferença de aplicação das estruturas de repetição *for*, *while* e *do-while*?
2. Faça um código que conte até 10 no console.
3. Faça um código que conte até 20, de 2 em 2.
4. Faça um código que conte regressivamente de 10 a 0.
5. Considere dois inteiros, a e b. Faça um programa que encontre a partir de qual o valor de a, quando elevado ao quadrado, será maior que b.
6. Teste no programa os dois códigos abaixo. Perceba que eles têm o mesmo critério de parada. O que ocorre? Por que os resultados são diferentes?

Código 1:

```
public class Main {
    public static void main(String[] args) {
        int i = 0;
        while (i != 0){
            System.out.println(i);
            i++;
        }
        System.out.println(i);
    }
}
```

Código 2:

```
public class Main {
    public static void main(String[] args) {
        int i = 0;
        do {
            System.out.println(i);
            i++;
        } while (i != 0);
        System.out.println(i);
    }
}
```

Chapter 6

Classes e Interfaces Importantes em Java

Working on it :)

6.1 Classes

Working on it :)

6.1.1 Importando Classes

Frequentemente enfrentamos problemas em programação que já foram solucionados por outros programadores. Linguagens como Python e R utilizam muito o conceito de pacotes na programação. Por exemplo, se você pretende criar gráficos em R dificilmente você irá criar ele desde o início, mas usará um pacote já existente. Neste caso, o pacote `ggplot2` é o mais utilizado.

A linguagem Java também tem a função de importar classes já existentes para facilitar a programação. Conhecer as principais classes e suas aplicações são muito úteis. Para que seja possível utilizar as classes, é necessário inicialmente importa-la. Por padrão, as primeiras linhas dos códigos (antes de serem instanciadas as classes) são utilizadas para essa finalidade.

```
//Espaço para importar classes

public class Main {
    public static void main(String[] args){
        System.out.println("Hello World!");
```

```
    }
}
```

O código para importar classes é imposto por `"import + nome do pacote + nome da classe"`. Diferentemente das linguagens Python e R, é necessário que esses pacotes já estejam no ambiente de desenvolvimento Java pois a linguagem não busca os pacotes diretamente da internet.

Um caso de pacote frequentemente utilizado é o `java.util`, que contém classes extremamente utilizadas como `ArrayList` e `Random`. Para importar esse pacote, basta digitar o código abaixo no local já indicado.

```
import java.util.ArrayList; //Importa apenas uma classe do pacote java.util
import java.util.*; // Importa todas as classes do pacote java.util
```

Note que na segunda linha de código não foi especificado o nome da classe. Isso quer dizer que na primeira linha apenas a classe `ArrayList` está sendo importada, enquanto na segunda todas as classes (incluindo a `ArrayList`) do pacote `java.util` estão sendo importadas.

6.2 Interfaces

Working on it :)

6.2.1 Comparable

Classe para ordenar objetos.

6.2.1.1 Aplicando

Passo 1: implementar a interface `Comparable` na classe com base no código abaixo.

```
public class Classe implements Comparable<Classe>
```

Passo 2: Reescrevendo o método `compareTO(Classe objeto)` como no exemplo abaixo.

```
@Override
public int compareTo(Individual individual) {
```

```
    double compareQuantity = 100*((Individual) individual).getCost();
    double thisCost = 100*this.cost;

    //ascending order
    return (int) Math.round(thisCost - compareQuantity);
}
```

Chapter 7

Vetores, Matrizes e Listas

Muitas vezes em programação temos que trabalhar com conjuntos de valores em conjunto. Para isso é possível utilizarmos as ferramentas de vetores e listas. Esses conjuntos têm diversas semelhanças e diferenças entre si. Isso impacta diretamente na escolha de qual delas deve ser utilizada no código. Aqui serão apresentadas cada uma delas e suas particularidades.

Imagine uma fila. Nela existem várias pessoas em uma determinada sequência. Há posições nessa fila e as pessoas estão ordenadas de acordo com uma ordem específica (por nome, idade, ordem de chegada, enfim, qualquer ordem). Assim funcionam os vetores. Os vetores funcionam como "filas" onde cada elemento está posicionado em uma determinada posição.

Em Java, diferentemente de outras linguagens de programação (R, por exemplo), a contagem de posições começa no 0. Ou seja, um vetor com 10 posições tem sua primeira posição em 0 e sua última posição em 9 (contabilizando, assim, posições). Uma representação gráfica é apresentada abaixo.

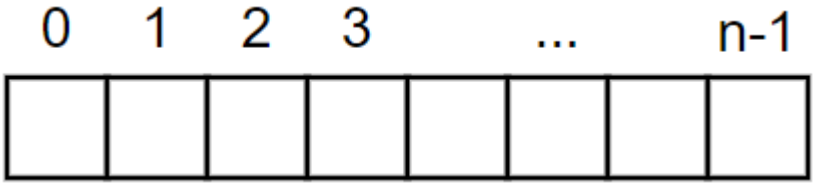

Figure 7.1: Vetores

7.1 Arrays

As *arrays* são objetos que armazenam valores em sequência.Cada valor em uma *array* é chamado de **elemento** Esse tipo de vetor tem um número fixo de posições que é determinado no momento de sua criação. Eles podem armazenar qualquer tipo de variável ou objeto. A estrutura de código para criar uma *array* é apresentada abaixo.

```
tipo[ ] nome = new tipo[tamanho]
```

Diversos pontos devem ser considerados para criar uma *array*:

- Toda *array* deve ter um tipo. Isso significa que todos os elementos da *array* deverão ser do mesmo tipo, e.g., uma array do tipo `int[ ]` receberá apenas valores inteiros;
- Toda *array* depende de um tamanho. Esse tamanho é determinado no momento da criação da *array* por meio do termo `tamanho`.

Abaixo temos alguns exemplos de `arrays`.

```
int[] inteiros1 = new int[5];
double[] decimais = new  double[7];
String[] strings = new String[10];
float[] floats = new float[3];
```

Todas as *arrays* acima tem um tamanho determinado (os tamanhos são 5, 7, 10 e 3, respectivamente). Há uma forma de se criar uma *array* com todos os valores, conforme apresentado abaixo.

```
int[] inteiros2 = {0,1,2,3,4};
```

Da mesma forma que o vetor `inteiros1`, a *array* `inteiros2` tem 5 posições. A diferença de uma forma para a outra é que, a primeira *array* não possui valores pré-alocados, enquanto da segunda as posições já contam com valores pré-determinados.

7.1.1 Acrescentando ou Mudando Valores

Para acrescentar ou modificar um valor em uma posição deve-se indicar exatamente para qual posição cada valor deve ser alocado.

```
int[] inteiros = {0,1,2,3,4};
inteiros[0] = 10;
// inteiros == {10,1,2,3,4}
```

No exemplo acima, o valor na posição 0 (de valor 0) foi substituído por 10.

7.1.2 Acessando Valores

De forma semelhante, para acessar um valor em uma *array* basta indicar a posição desejada.

```
int[] inteiros = {0,1,2,3,4};
int inteiro = inteiros[3];
//inteiro == 3;
```

7.1.3 Valores e Referências

Corra o código abaixo. O que acontece?

```
public class Main {

    public static void main(String[] args){

        int[] inteiros1 = {48, 9, 96, 26, 10, 51};
        int[] inteiros2 = inteiros1;
        System.out.println("inteiros1 =  " + Arrays.toString(inteiros1));
        System.out.println("inteiros2 =  " + Arrays.toString(inteiros2));

        inteiros2[0] = 47;

        System.out.println("inteiros1 =  " + Arrays.toString(inteiros1));
        System.out.println("inteiros2 =  " + Arrays.toString(inteiros2));
    }
}
```

O resultado que aparece no console deve ser igual ao apresentado abaixo.

```
inteiros1 =  [48, 9, 96, 26, 10, 51]
inteiros2 =  [48, 9, 96, 26, 10, 51]
inteiros1 =  [47, 9, 96, 26, 10, 51]
inteiros2 =  [47, 9, 96, 26, 10, 51]
```

Mas, em contra partida, se correrms o código abaixo:

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

        int numero1 = 10;
        int numero2 = numero1;

        System.out.println("numero1 = " + numero1);
        System.out.println("numero2 = " + numero2);

        numero2++;

        System.out.println("numero1 = " + numero1);
        System.out.println("numero2 = " + numero2);
    }
}
```

O resultado será:

```
numero1 = 10
numero2 = 10
numero1 = 10
numero2 = 11
```

Qual a diferença? Por que em um caso mudou os dois objetos enquanto no outro mudou apenas um? A explicação por trás desse fenômeno é bem simples: Referências.

A forma como o sistema trabalha com variáveis e arrays é diferente. Quando trabalhamos com variáveis o computador aloca um espaço para cada variável. Dessa forma, cada variável criada tem um espaço próprio e independente de outras variáveis na memória.

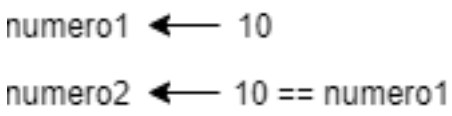

Figure 7.2: Variáveis

Em contrapartida o computador não cria um no espaço para salvar o novo vetor quando criamos uma nova *array*. Dessa forma, estamos apenas abrindo dois caminhos para encontrarmos o mesmo objeto.

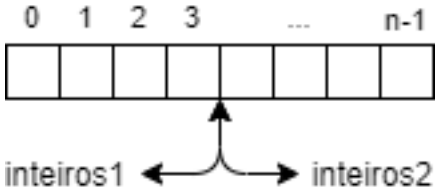

Figure 7.3: Arrays

7.1.3.1 Copiando Arrays sem Referência

Felizmente, o Java traz um método muito prático para solucionar este problema. Para tanto, precisamos importar a classe `Arrays`. O método é chamado `copyOf` e é utilizado como mostrado abaixo.

```
import java.util.Arrays;
public class Main {

    public static void main(String[] args){
        int[] vetor1 = {02, 06, 18, 28, 28};
        int[] vetor2 = Arrays.copyOf(vetor1, vetor1.length);

        System.out.println("vetor1 = " + Arrays.toString(vetor1));
        System.out.println("vetor2 = " + Arrays.toString(vetor2));

        vetor2[4] = 29;

        System.out.println("vetor1 = " + Arrays.toString(vetor1));
        System.out.println("vetor2 = " + Arrays.toString(vetor2));

    }
}
```

E o resultado será:

```
vetor1 = [2, 6, 18, 28, 28]
vetor2 = [2, 6, 18, 28, 28]
vetor1 = [2, 6, 18, 28, 28]
vetor2 = [2, 6, 18, 28, 29]
```

7.2 Matrizes

Em Java as matrizes são vistas como vetores de vetores (ou *arrays* de *arrays*). Dessa forma, As matrizes não necessitam ter as linhas de mesmo tamanho. Por exemplo, no código abaixo temos uma matriz de 3 linhas com tamanhos diferentes (5,3 e 1 respectivamente).

```
import java.util.Arrays;
public class Main {

    public static void main(String[] args){

        int[][] matriz = new int[3][];
        matriz[0] = new int[5];
        matriz[1] = new int[3];
        matriz[2] = new int[1];

        for(int i = 0; i < matriz[0].length; i++) {matriz[0][i] = i + 1;}
        for(int i = 0; i < matriz[1].length; i++) {matriz[1][i] = i + 6;}
        for(int i = 0; i < matriz[2].length; i++) {matriz[2][i] = i + 9;}

        System.out.println("Matriz");
        for(int i = 0; i < matriz.length; i++) {System.out.println(Arrays.toString(mat

    }
}
```

Tendo como resultado:

```
Matriz
[1, 2, 3, 4, 5]
[6, 7, 8]
[9]
```

Perceba que a mudança na sintaxe para a criação de uma matriz é muito semelhante à criação de uma *array*.

```
int[][] matriz = new int[numeroLinhas][];
```

A única informação necessária de ser incluída é o número de linhas. É possível também deixar pré-determinado o número de colunas no segundo colchetes.

```
int[][] matriz = new int[numeroLinhas][numeroColunas];
```

Dessa forma a matriz se mantém no formato clásico, conforme apresentado abaixo.

```
import java.util.Arrays;
public class Main {

    public static void main(String[] args){
        int[][] matriz = new int[3][6];

        System.out.println("Matriz");
        for(int i = 0; i < matriz.length; i++) {System.out.println(Arrays.toString(matriz[i]));}

    }
}
```

De resultado:

```
Matriz
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
```

7.3 Listas

Em Java existe uma interface própria para listas chamada `List`. As listas são vistas como uma coeção ordenada (também conhecido como *sequência*). Utilizar essa interface dá um controle preciso sobre os elementos inseridos. As classes mais comuns de se trabalhar como listas são as `ArrayLists` e as `LinkedLists` que serão melhor explicadas nas subseções seguintes.

7.3.1 ArrayLists

Parágrafo Introdutório

Criando ArrayLists

Estrutura para criar uma ArrayList.

```
ArrayList<TIPO> nome = new ArrayList<>();
```

Código para criar uma ArrayList

```
import java.util.ArrayList;

public class Main {
    ArrayList<Integer> arraylist = new ArrayList<>();
}
```

Adicionando valores a uma ArrayList

Estrutura para adicionar um valor a uma ArrayList.

```
nome.add(índice, valor);
```

Exemplo.

```
import java.util.ArrayList;
import java.util.Arrays;

public class Main {
    public static void main(String[] args){
        ArrayList<Integer> arraylist1 = new ArrayList<>();
        arraylist1.add(9);
        arraylist1.add(10);
        System.out.println("ArrayList = " + Arrays.toString(arraylist1.toArray()));
        arraylist1.add(0,8);
        System.out.println("ArrayList = " + Arrays.toString(arraylist1.toArray()));
    }
}
```

O resultado impresso no console será, então.

```
ArrayList = [9, 10]
ArrayList = [8, 9, 10]
```

Modificando valores de uma ArrayList

Estrutura do código.

```
nome.set(índice, valor);
```

Exemplo de aplicação.

```
import java.util.ArrayList;
import java.util.Arrays;

public class Main {
    public static void main(String[] args){
        ArrayList<Integer> arraylist1 = new ArrayList<>();
        arraylist1.add(1);
        arraylist1.add(3);
        arraylist1.add(3);
        System.out.println("ArrayList = " + Arrays.toString(arraylist1.toArray()));
        arraylist1.set(1,2);
        System.out.println("ArrayList = " + Arrays.toString(arraylist1.toArray()));
    }
}
```

Imprimindo o seguinte resultado.

```
ArrayList = [1, 3, 3]
ArrayList = [1, 2, 3]
```

Selecionando valores de uma ArrayList

Estrutura do código.

```
nome.get(posição)
```

Exemplo

```
import java.util.ArrayList;
public class Main {
    public static void main(String[] args){
        ArrayList<Integer> arraylist1 = new ArrayList<>();
        arraylist1.add(1);
        arraylist1.add(2);
        arraylist1.add(3);
        System.out.println("Valor = " + arraylist1.get(1));
    }
}
```

7.3.1.1 Tamanho de uma ArrayList

Código para saber o tamanho de uma ArrayList.

```
nome.size();
```

Aplicação.

Exercícios

Exercícios com Arrays

1. Crie uma *array* de inteiros com 5 posições.
2. Crie uma *array* de decimais com 7 posições com valores pré-definidos (a sua escolha).
3. Crie uma *array* de *strings* com 3 posições sem valores pré-definidos. Após a criação, inclua valores para cada uma das posições.
4. Crie uma *array* de decimais com 7 posições com valores pré-definidos (a sua escolha). Faça alteração dos valores de 3 posições (a sua escolha).
5. Crie uma *array* de decimais com 7 posições com valores pré-definidos (a sua escolha). Imprima os valores dessa array na tela de comando de 2 formas diferentes. (Dica: utilize um método da classe `Arrays` para uma forma e utilize um `for()` para a outra).
6. Crie uma *array* de decimais com 7 posições com valores pré-definidos (a sua escolha). Crie uma outra *array* copiando os valores da primeira sem que haja referência à mesma *array* de 2 formas diferentes. (Dica: utilize um método da classe `Arrays` para uma forma e utilize um `for()` para a outra).

Exercícios com Matrizes

1. Crie uma matriz de inteiros com 4 linhas.
2. Crie uma matriz de inteiros com 4 linhas e 5 colunas.
3. Imprima no console as matrizes do exercício anterior de duas maneiras diferentes. (Dica: utilize um método da classe `Arrays` para uma forma e utilize um `for()` para a outra).

Chapter 8

Coleções

Working on it :)

8.1 Maps

Working on it :)

8.2 Sets e HashSets

Working on it :)

8.3 StockList

Chapter 9

Tópicos Avançados em Java

Working on it :)

9.1 Classes, Construtores e Herança

Working on it :)

9.2 Composição, Encapsulamento e Polimorfismo

Working on it :)

9.3 Classes Abstratas e Interfaces

Chapter 10

Pesquisa Operacional e Solução de Problemas

Working on it :)

10.1 Heurísticas e Meta-Heurísticas

Working on it :)

Chapter 11

Aplicações em Logística e Cadeia de Suprimentos

Working on it :)

11.1 Cadeia de Suprimentos Varejista

Working on it :)

11.2 Integração de Fornecedores, Produção e Distribuição

Working on it :)

Chapter 12

Problemas Clássicos de Computação

Working on it :)

Considerações Finais

Working on it :)

Respostas dos Exercícios

Working on it :)