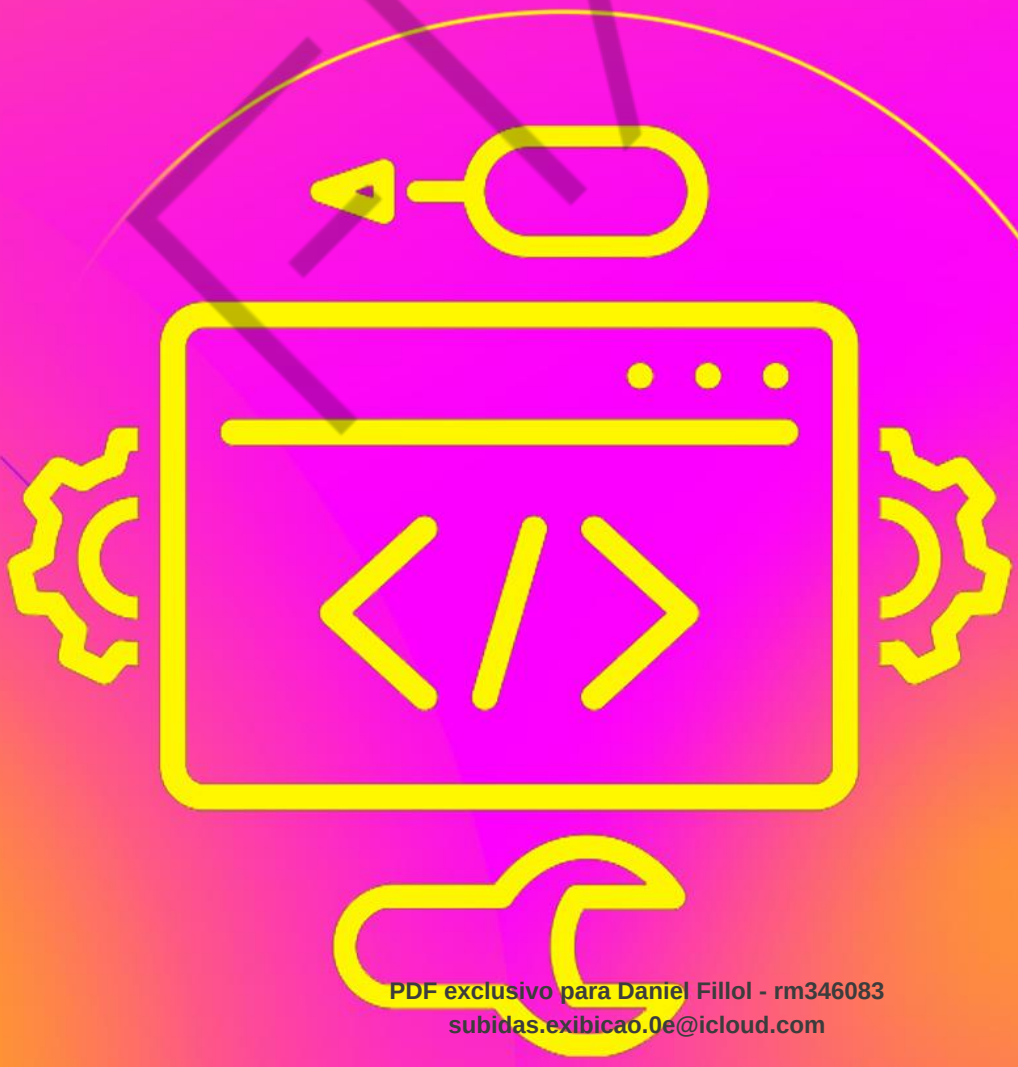
FIAP ON

FUNDAMENTOS JAVA

ESTRUTURAS

THIAGO T. I. YAMAMOTO



5

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

5 ESTRUTURAS

Estruturas de repetição ou loops permitem que um bloco de código seja executado repetidamente, enquanto alguma condição permanecer verdadeira.

No Java existem três estruturas de repetição que veremos a seguir: While, Do-While e For.

5.1 While

O loop *while* executa um bloco de código enquanto a condição for verdadeira.

Estrutura básica do comando:

```
while (<condição>){
    <Instruções>
}
```

Perceba que o loop *while* nunca será executado se a condição for *false* desde o início.

O exemplo abaixo exibe um trecho de código que imprime os números do 1 ao 10:

```
int numero = 0;
while (numero < 10){
    numero = numero + 1 ;
    System.out.println(numero);
}
```

No exemplo acima, o número é iniciado com o valor 0. Na primeira iteração, ele é menor do que 10, então o *while* é executado. O número é incrementado em uma unidade e seu valor é exibido no console. Na segunda iteração, o número possui o valor 1, assim o loop será executado novamente.

Quando o número for 10, o loop será executado pela última vez, pois o número será incrementado novamente e a condição do *while* será falsa, pois o número será 11, ou seja, maior do que 10.

O loop *while* é testado na parte inicial do loop, antes que seja executada a primeira iteração. Outra forma, seria movendo esse teste para o final do bloco, assim garantimos que o bloco de código será executado ao menos uma vez. Desta forma, é utilizada outra estrutura de repetição chamada *do-while*, que veremos a seguir.

5.2 Do-While

Esse loop primeiramente executará todo o bloco de código para depois testar a condição e assim verificar se repete novamente o bloco de código.

Sintaxe básica:

```
do {
    <instruções>
} while(<condição>);
```

Vamos desenvolver um exemplo com o mesmo funcionamento do exemplo anterior.

```
int numero = 0;
do {
    numero = numero + 1;
    System.out.println(numero);
} while(numero < 10);
```

A única diferença entre os loops while e do-while é a posição do teste para a repetição.

Os loops acima são mais utilizados quando não sabemos exatamente quantas repetições serão efetuadas. Nesses exemplos, sabemos que cada loop será executado 10 vezes. Para esse tipo de repetição, existe um outro loop, mais indicado, que será abordado em breve.

Agora, vamos montar um exemplo mais interessante para o nosso loop:

```
Scanner sc = new Scanner(System.in);
String opcao;
do {
    System.out.println("Digite um número");
    int n1 = sc.nextInt();
    System.out.println("Digite outro número");
    int n2 = sc.nextInt();
    int soma = n1 + n2;
    System.out.println("A soma é  " + soma);
    System.out.println("Deseja somar outro número? (S/N)");
    opcao = sc.next();
} while(opcao.equals("S"));
sc.close();
```

Neste exemplo, primeiro leremos os dois números inseridos pelo usuário para efetuar a soma. Após, verificaremos se o usuário deseja realizar outra soma. Caso o usuário digite “S”, o loop será executado novamente (lembre-se de que para verificar a igualdade de *strings,* é necessário utilizar o método equals ao invés do operador ==).

Para esse exemplo, utilizaremos o *do-while,* pois é necessário que o bloco de código seja executado ao menos uma vez. Outro ponto importante do exemplo é que não podemos definir inicialmente quantas vezes o loop será executado, pois isso dependerá dos valores inseridos pelo usuário.

A última estrutura de repetição que será abordada é o *for,* indicado quando sabemos exatamente quantas vezes vamos repetir o loop*,* ou seja, quando ela é controlada.

5.3 For

O loop *for* é uma estrutura de repetição controlada por uma variável de contador que será atualizada depois de cada iteração.

Sintaxe básica:

```
for (<inicialização>;<condição lógica>;<incremento ou decremento>){
    <instruções>
}
```

O primeiro valor (<inicialização>) da estrutura for normalmente é utilizado para inicializar a variável de contador. A condição lógica verifica se o loop deve repetir ou não, e o incremento (ou decremento) é a atualização da variável de controle, a cada iteração. Por exemplo:

```
for (int i=0; i<=10; i++){
    System.out.println(i);
}
```

Este exemplo imprime os valores de 0 a 10. Observe que a atualização da variável de controle foi realizada dentro do *for*, assim, não é necessária a sua atualização dentro do bloco de código (apesar de ser possível modificá-la dentro do bloco de código também).

Como imprimiremos os valores de 0 a 10, a condição lógica utilizada foi para repetir enquanto a variável *i* for menor ou igual a 10. Assim, quando a variável *i* for igual a 10, ela será exibida no console e atualizada para 11, e depois o loop não será mais executado.

5.4 *Arrays*

Até o momento, nós trabalhamos com variáveis que armazenam uma informação. Imagine agora se fosse necessário armazenar 40 itens do mesmo tipo, como por exemplo, armazenar todas as notas dos alunos de uma disciplina. Poderíamos criar 40 variáveis diferentes, porém essa é uma forma muito trabalhosa e propensa a erros. Imagine agora se fossem 100 alunos?

Um *array* é uma estrutura de dados que armazena uma coleção de itens do mesmo tipo, que pode ser um tipo primitivo ou um objeto. Cada item no vetor possui o seu próprio local numerado, chamado índice. O índice é utilizado para acessar um elemento no vetor e, assim, recuperar ou atribuir uma informação naquele índice.

Em um *array* o índice se inicia a partir do 0, ou seja, para acessar o elemento que está na segunda posição do *array* utilizaremos o índice 1 e assim sucessivamente.

Um *array* possui um comprimento fixo e que não pode ser alterado, ou seja, será definido um valor para o *array* e esse deve permanecer. Existe a possibilidade de recuperar o tamanho do *array* por meio do atributo **length.** A figura 1 apresenta um array com um tamanho de 10 posições e seus índices respectivos:

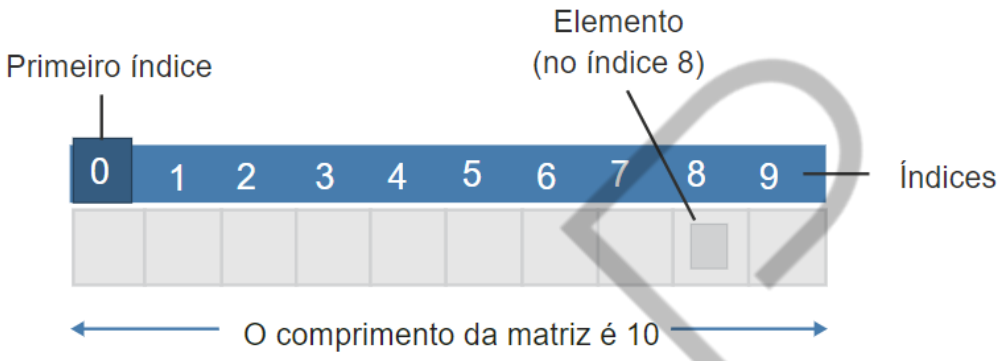


Figura 5.1 – Vetor com 10 posições
Fonte: Elaborado pelo autor (2019)

Um array em Java é um objeto. Dessa forma, podemos utilizar o operador ***new*** para criar uma nova instância de um *array.*

Para declarar uma variável para armazenar um *array,* primeiro precisamos especificar o tipo do *array*, acrescentar colchetes ([ ]) e definir o nome da variável. Exemplo:

```
int[] notas;
```

Neste exemplo, nós declaramos a variável notas, a qual armazena um conjunto de números inteiros. Também é possível adicionar os colchetes depois do nome da variável:

```
int notas[];
```

As duas formas produzem o mesmo resultado. Agora, podemos inicializar um *array* utilizando o operador **new:**

```
int[] notas = new int[40];
```

No momento da declaração, precisamos definir entre os colchetes o comprimento do *array*. Neste exemplo, criamos um array de 40 números inteiros. Ao criar um *array* de número, todos os elementos são inicializados com 0. Um *array* de boolean é inicializado com *false* e um *array* de objeto é inicializado com *null.*

Para atribuir um valor em uma posição do *array*, utilizamos o índice dentro dos colchetes. No exemplo abaixo, estamos armazenando na primeira posição do vetor o número 10:

```
notas[0] = 10;
```

E para recuperar o valor, basta dizer o índice do vetor:

```
System.out.println(notas[0]);
```

No exemplo acima, recuperamos o valor armazenado no índice 0 do vetor exibimos no console.

Existem duas formas de declarar um vetor com suas posições preenchidas com valores pré-determinados:

A primeira, consiste em atribuir os valores entre chaves { } e separado por vírgula:

```
int notas[] = {10,9,8,2};
```

No exemplo acima foi declarado um vetor com tipo de dado inteiro e com 4 posições.

A outra forma tem o mesmo princípio da primeira, diferenciando pela adição do operador **new** na declaração:

```
int notas[] = new int[]{10,9,8,2};
```

O resultado das declarações acima é igual ao resultado do código abaixo:

```
int notas[] = new int[4];
notas[0] = 10;
notas[1] = 9;
notas[2] = 8;
notas[3] = 2;
```

Podemos criar *arrays* de qualquer tipo de dado no Java: string, byte, char, int, long, double, float, boolean ou qualquer classe Java.

Exemplos:

```
byte bytes[] = new byte[4];
short shorts[] = new short[8];
double doubles[] = new double[7];
float floats[] = new float[3];
String strings[] = new String[10];
Carro carros[] = new Carro[15];
```

Agora, como podemos acessar todas as posições de um vetor?

A resposta são os loops. Eles são perfeitos, pois podemos fazer um loop para repetir uma quantidade de vezes igual ao tamanho do vetor.

Podemos pensar em outro exemplo, imagine um programa que precisa armazenar as notas de uma turma de 10 alunos e calcular a sua média. Primeiro, vamos ler as notas dos 10 alunos utilizando um loop. Depois vamos calcular a média e exibir para o usuário.

No exemplo abaixo, foi instanciado o scanner para realizar a leitura do teclado. Depois, criamos um vetor de float com 10 posições. Assim, vamos utilizar a estrutura de repetição for para ler e armazenar em cada posição do vetor as notas dos alunos.

```
Scanner sc = new Scanner(System.in);
float[] notas = new float[10];
for (int i = 0; i < 10; i++) {
    System.out.println("Digite a nota do aluno");
    notas[i] = sc.nextFloat();
}
sc.close();
```

Observe que a variável *i* é utilizada como um índice, para controlar a posição do vetor que será utilizado para armazenar o valor. Dessa forma, quando o programa for executado pela primeira vez, na primeira iteração do loop o valor de *i* será 0 que é a primeira posição do vetor. Já na última vez que o loop for executado, o valor de *i* será 9 que é a última posição do vetor. Lembrando que em Java o vetor inicia sempre na posição 0, dessa forma, como o vetor tem 10 posições, ele inicia no 0 e finaliza em 9.

Podemos otimizar nosso código, ao invés de deixar o valor 10 fixo no comando *for;* primeiramente, vamos recuperar o tamanho do vetor e utilizar esse valor recuperado para ser utilizado como critério de parada do loop.

Dessa forma, não precisaremos modificar o loop se o vetor possuir mais ou menos posições.

```
for (int i = 0; i < notas.length ; i++)
```

Agora vamos desenvolver a segunda parte do programa: calcular a média de notas. Para isso, vamos utilizar outro loop para somar todas as notas dos alunos, e depois dividir pela quantidade de notas, que neste caso é 10.

```
// Calcular a média dos alunos
float totalNotas = 0;
for (int i = 0; i < notas.length; i++) {
    totalNotas = totalNotas + notas[i];
}
float mediaNotas = totalNotas/notas.length;
System.out.println("A média dos alunos é " + mediaNotas);
```

Na primeira linha, declaramos uma variável do tipo float para armazenar a soma de todas as notas dos alunos. Depois implementamos um loop para percorrer todo o vetor e ir somando as notas na variável totalNotas.

Por fim, realizamos a divisão do total de notas pela quantidade de alunos, para calcular a média e imprimimos no console o resultado para o usuário. Perceba que para o loop e para a divisão, utilizamos o valor da quantidade de elementos do vetor.

No exemplo acima, utilizamos um *array* para armazenar números primitivos. Já falamos que é possível criar um *array* para armazenar objetos, que são os *arrays* de referências ou o "*array* de objetos". Neste *array* é possível armazenar várias referências a um tipo de objeto.

Neste exemplo, estamos definindo um array com 5 posições para armazenar as referências de objetos do tipo Carro.

```
Carro[] carros = new Carro[5];
```

Até este momento, nenhum objeto Carro foi criado. O *array* foi criado para armazenar as referências de 5 Carros, e por enquanto as posições do vetor estão vazias (*null*).

Para popular um *array* de carros, primeiro é preciso instanciá-lo, e depois armazenar a sua referência em uma posição do vetor:

```
Carro[] carros = new Carro[5];

Carro carro = new Carro();
carro.setModelo("Gol");

carros[0] = carro;
```

Assim, a posição 0 do vetor possui uma referência a um objeto Carro. As outras posições ainda estão vazias. A figura a seguir representa o estado atual do *array* de carros:

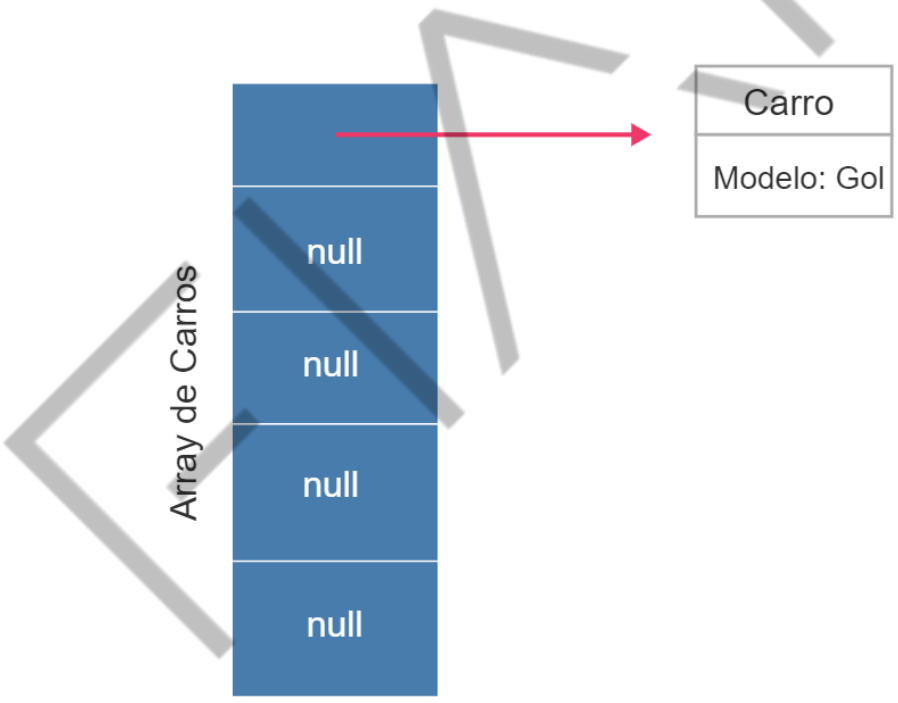


Figura 5.2 – Estado atual do array de carros
Fonte: Elaborado pelo autor (2019)

Para recuperar o valor do modelo do carro que está armazenado na primeira posição do vetor, podemos escrever o seguinte código:

```
String modelo = carros[0].getModelo();
System.out.println(modelo);
```

Este código recupera o valor do modelo do carro e imprime no console do eclipse.

Podemos percorrer o vetor de referências da mesma forma que fizemos com o *array* de primitivos, utilizando as estruturas já vistas neste capítulo. Porém, além dessas estruturas, é possível percorrer um vetor de primitivos ou referência com uma sintaxe mais simples.

Estamos falando do comando **for-each,** que não necessita manter uma variável de controle para indicar a posição do elemento no vetor.

A sintaxe é:

```
for (<Tipo> <variável> : <Array>) {
}
```

O primeiro parâmetro é o tipo do *array*. O segundo é um nome para a variável que vai receber cada um dos itens do vetor. O último parâmetro, que está após os dois pontos (:) é o *array* que queremos percorrer.

Imagine o vetor de carros acima. Podemos percorrê-lo com o seguinte código:

```
for (Carro carro : carros) {
    System.out.println(carro.getModelo());
}
```

O tipo do vetor é **Carro**, a variável que vai receber cada um dos itens do vetor é o **carro** e o *array* que queremos percorrer é o vetor de **carros**.

Para um vetor de primitivos, como o *array* de notas que desenvolvemos acima, também podemos utilizar o **for-each**:

```
for (float nota : notas) {
    System.out.println(nota);
}
```

Os *arrays* que acabamos de ver são os vetores unidimensionais. Existem também as matrizes ou arrays multidimensionais. As matrizes nada mais são do que *arrays* de *arrays*. Dessa forma, cada posição do array armazena outro *array.* Esses arrays também podem conter *arrays*, e assim por diante, ou seja, quantas dimensões que o desenvolvedor desejar.

Imagine agora que as notas dos alunos devem ser armazenadas por disciplinas. O curso tem 9 disciplinas com 40 alunos cada. Dessa forma, podemos criar um *array* com 9 posições e em cada posição armazenar um outro *array* com 40 elementos:

```
int[][] notas = new int[9][40];
```

O *array* de *array* denominado notas tem 360 posições, uma para cada aluno em 9 disciplinas. Você pode armazenar a nota do primeiro aluno para a primeira disciplina com a seguinte instrução:

```
notas[0][0] = 10;
```

Para acessar a nota do segundo aluno da primeira disciplina, basta alterar o índice:

```
notas[0][1] = 9;
```

Portanto, os *arrays* de *arrays* funcionam da mesma forma que os *arrays* unidirecionais. O primeiro índice de cada *array* começa em 0.

É possível também criar um *array* de *array* de *array* ou em quantas dimensões forem necessárias:

```
int[][][] notas = new int[10][50][10];
```

5.5 *Strings*

Strings nada mais são do que uma sequência de caracteres. Durante o curso, já utilizamos *strings* para armazenar palavras e textos. Agora é o momento de aprender a manipular as *strings*, pois é essencial para fazer validações de dados de entrada, exibir informações para o usuário e outras informações baseadas em texto.

O Java não tem um tipo de dado primitivo como int ou double para armazenar uma *string*. Ao invés, podemos utilizar a biblioteca padrão Java que contém uma classe predefinida chamada **string**.

Os objetos *strings* são imutáveis, isto é, seu conteúdo de caracteres não pode ser alterado após a sua inicialização. Dessa forma, é impossível alterar o valor da *string*. Porém, é possível armazenar outra *string* no lugar da *string* original.

Uma *string* deve ser declarada, instanciada e inicializada.

Nós já declaramos uma *string* durante o curso. Declarar uma *string* é igual a declarar uma variável ou atributo de qualquer outro tipo de dado, basta declarar o tipo de dado e o nome da variável, conforme exemplo abaixo:

```
String nome;
```

Neste caso, o tipo de dado utilizado é **string** e o nome da variável é **nome**. Depois de declarar uma *string*, podemos instanciá-la como uma classe normal. A sintaxe para a instanciação é:

```
String nome = new String();
```

É possível inicializar uma *string* de diversas formas. Podemos instanciá-la e depois atribuir um valor ou atribuir um valor no momento da instanciação:

```
String nome = new String();
nome = "FIAP";
```

ou

```
String nome = new String("FIAP");
```

Neste último exemplo, utilizamos um construtor da classe *string* para passar um valor. É possível também atribuir um valor a uma *string* sem instanciá-la.

```
String nome = "FIAP";
```

Dessa forma, a *string* será armazenada em um *pool* de *strings*, uma área utilizada pelo Java como cache.

Caso a *string* seja uma variável de instância, ou seja, um atributo de uma classe, ela precisa ser instanciada ou inicializada antes de ser utilizada. Pois todos os atributos de referência são inicializados com *null.* Dessa forma,

se invocarmos um método de uma variável vazia, vamos receber uma exception chamada NullPointerException. Não se preocupe, veremos exceções no decorrer do curso. Neste momento, o importante é entender que um erro pode acontecer.

No exemplo abaixo, estamos utilizando o método length() para recuperar a quantidade de caracteres da *string* nome sem instanciá-la:

```
String nome = null;
System.out.println(nome.length());
```

A variável nome acima não possui nenhum valor, ou seja, está com *null*, quando o programa executar a linha que recupera o tamanho da *string* e tentar imprimi-la no console, um erro ocorrerá.

Para corrigirmos e esse erro não acontecer, é preciso instanciar ou inicializar uma *string*:

```
String nome = new String();
System.out.println(nome.length());
```

ou

```
String nome = "FIAP";
System.out.println(nome.length());
```

Lembrando que quando declaramos uma variável dentro de um método é preciso sempre inicializá-la antes de utilizá-la. Caso isso não aconteça, um erro de compilação será exibido:

```
String nome;
System.out.println(nome.length());
```

No exemplo acima, o código não compila, pois estamos utilizando a variável nome antes de atribuir um valor a ela.

Agora que entendemos como uma *string* funciona, vamos pensar como podemos imprimir ou armazenar aspas (“) ou adicionar uma quebra de linha na *string*?

Para isso, podemos utilizar os caracteres de **escape** que são alguns caracteres precedidos da contra barra (\). Esses caracteres são considerados sequência de escape e têm um significado especial para o compilador.

No Java as sequências de escape mais utilizadas são:

Sequência de Escape	Descrição
\n	Nova linha. Posiciona o cursor no início da próxima linha.
\t	Tabulação horizontal. Move o cursor para a próxima posição da tabulação horizontal.
\\	Barras invertidas. Utilizada para gerar um caractere de barra invertida (\).
\"	Aspas duplas. Utilizada para gerar um caractere de aspas duplas.
\'	Aspas simples. Utilizada para gerar um caractere de aspas simples.

Quadro 5.1 – Sequências de Escape
Fonte: Elaborado pelo autor (2019)

Para utilizar uma sequência de escape basta adicioná-la a uma *string*:

```
String nome = "FIAP \nA melhor faculdade de tecnologia";
System.out.println(nome);
```

O resultado da execução acima será:

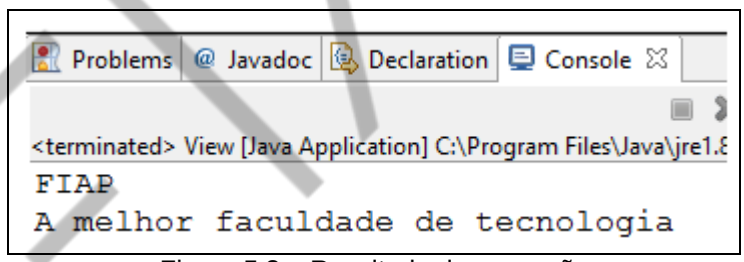


Figura 5.3 – Resultado da execução
Fonte: Elaborado pelo autor (2019)

É possível também adicionar as sequências de escape diretamente no método de impressão no console:

```
System.out.println("FIAP \nA melhor faculdade de tecnologia");
```

Teste as outras opções de sequência de escape. Por exemplo, adicione as aspas duplas em uma frase:

```
System.out.println("Faculdade: \"FIAP\"");
```

A saída no console será: **Faculdade: ”FIAP”**

Todos sabem que é possível somar números. E com as *strings,* também é possível?

Sim, também é possível “somá-las”, ou melhor, concatená-las. Concatenação de *strings* nada mais é do que juntar duas ou mais *strings* para criar uma nova *string*.

A forma mais fácil de concatenar uma *string* é utilizar o operador de soma (+). Veja o exemplo abaixo:

```
String nome = "FIAP";
String slogan = "A melhor faculdade de tecnologia";
String faculdade = nome + slogan;
System.out.println(faculdade);
```

Primeiro foram declaradas duas *strings* com valores “FIAP” e “A melhor faculdade de tecnologia”, respectivamente. Depois, declaramos uma terceira variável com o nome “**faculdade”** e atribuímos os valores das outras duas *strings*, concatenando-as. O resultado impresso no console será:

```
FIAPA melhor faculdade de tecnologia
```

As duas *strings* foram “somadas”, ou seja, foram ligadas. Porém, para ter um resultado melhor, podemos adicionar um espaço entre as *strings*:

```
String faculdade = nome + " " + slogan;
```

Dessa forma, concatenamos *três* strings: o valor da variável nome, a *string* com o valor de espaço e o valor da variável slogan. O resultado dessa vez será:

```
FIAP A melhor faculdade de tecnologia
```

Podemos concatenar qualquer quantidade de *strings* e como também adicionar as sequências de escape, conforme vimos acima.

Lembram-se do operador de atribuição aditiva (**+=**)? Que soma um valor e adiciona a própria variável da esquerda, exemplo:

```
int numero = 10;
numero +=10;
System.out.println(numero);
```

A variável número é inicializada com 10 e depois é somada a ela mais 10, resultado assim o valor de 20.

Podemos utilizar esse operador para *strings* também. Dessa forma, vamos concatenando *strings* e atribuindo o resultado para a mesma variável:

```
String nome = "FIAP";
String slogan = "A melhor faculdade de tecnologia";

String faculdade = nome;
faculdade += " - ";
faculdade += slogan;

System.out.println(faculdade);
```

O valor final armazenado na variável faculdade do exemplo acima é:

```
FIAP - A melhor faculdade de tecnologia
```

Além do operador **+** e **+=**, podemos concatenar *strings* utilizando o método **concat**:

```
String nome = "FIAP";
String slogan = "A melhor faculdade de tecnologia";
String faculdade = nome.concat(" - ").concat(slogan);
System.out.println(faculdade);
```

No exemplo acima, concatenamos o valor da variável nome com a *string* que apresenta um traço (-) e a *string* armazenada em slogan. O valor final da nova *string* foi armazenado na variável faculdade. Assim, o resultado é igual ao exemplo anterior:

```
FIAP - A melhor faculdade de tecnologia
```

É possível misturar as técnicas apresentadas para realizar a concatenação de *strings*:

```
String faculdade = nome.concat(" - ") + slogan;
```

Além do método concat, a classe string fornece diversas opções de métodos para a sua manipulação. Com eles, podemos comparar *strings*, acessar seus caracteres, pesquisar por um caractere ou uma sequência e transformar uma *string*. Veremos cada uma dessas opções a seguir:

5.5.1 Comparação de *strings*

A comparação de *strings* deve ser realizada através de métodos, cujos principais métodos para comparação são:

- **Equals(string)**: verifica a igualdade do valor das *strings*.
- **EqualsIgnoreCase(string)**: verifica a igualdade do valor das *strings* sem diferenciar as letras maiúsculas e minúsculas.

Não devemos utilizar o operador == para comparar *strings*, pois esse operador compara o endereço de memória que a *string* está alocada, ao invés do valor armazenado na *string*. Como no exemplo a seguir:

```
String nome = new String("FIAP");
String nome2 = new String("FIAP");
if (nome == nome2){
    System.out.println("As Strings são iguais");
}else{
    System.out.println("As Strings são diferentes");
```

As duas variáveis têm o mesmo valor, porém estão alocadas em endereços de memória diferentes. Dessa forma, o resultado será *"*As *strings* são diferentes*"*, pois o operador == compara o endereço de memória e não o valor.

A figura a seguir mostra que as variáveis **nome** e **nome2** têm um espaço de memória separado para armazenar o valor "FIAP", pois as *strings* foram instanciadas com o operador **new.**

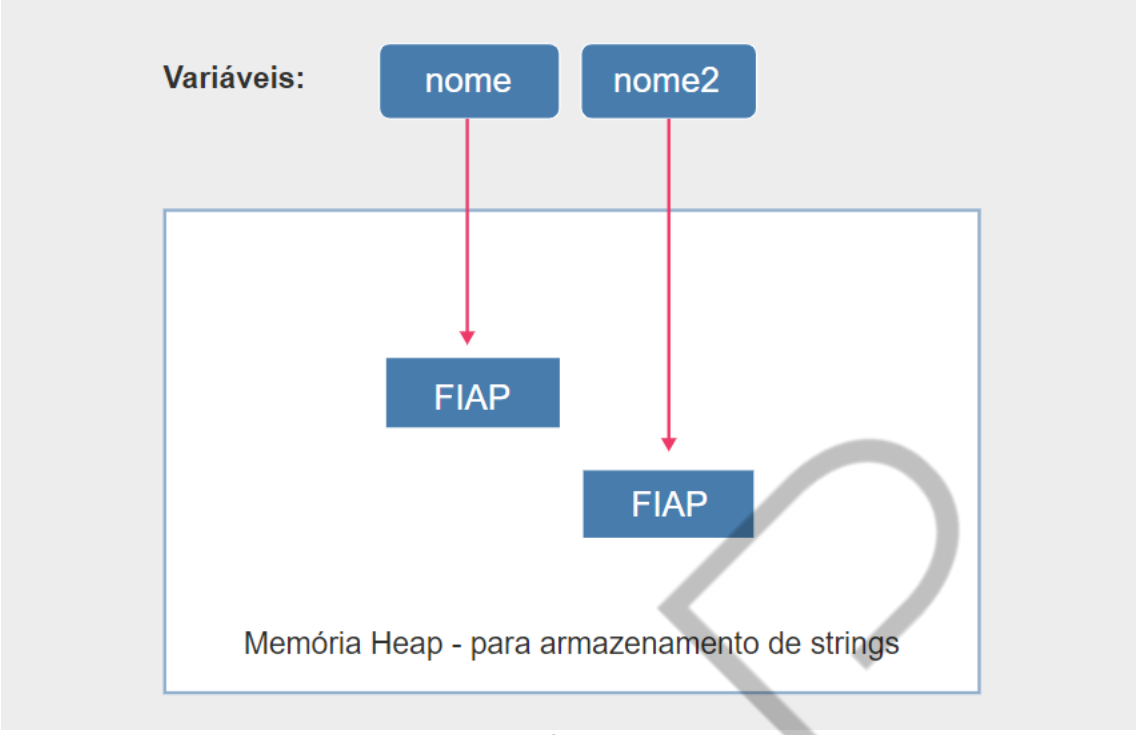


Figura 5.4 – Variáveis nome e nome2
Fonte: Elaborado pelo autor (2019)

Agora, se inicializar as *strings* sem instanciá-las, os valores serão alocados em um *pool* de *strings*. Dessa forma, se os valores forem iguais, elas vão compartilhar o mesmo espaço de memória no *pool*, fazendo com que o operador == funcione!

```
String nome = "FIAP";
String nome2 = "FIAP";
if (nome == nome2){
    System.out.println("As Strings são iguais");
}else{
    System.out.println("As Strings são diferentes");
}
```

O resultado é *"*As *strings* são iguais*"*, pois a variável **nome** e **nome2** compartilham o mesmo endereço de memória:

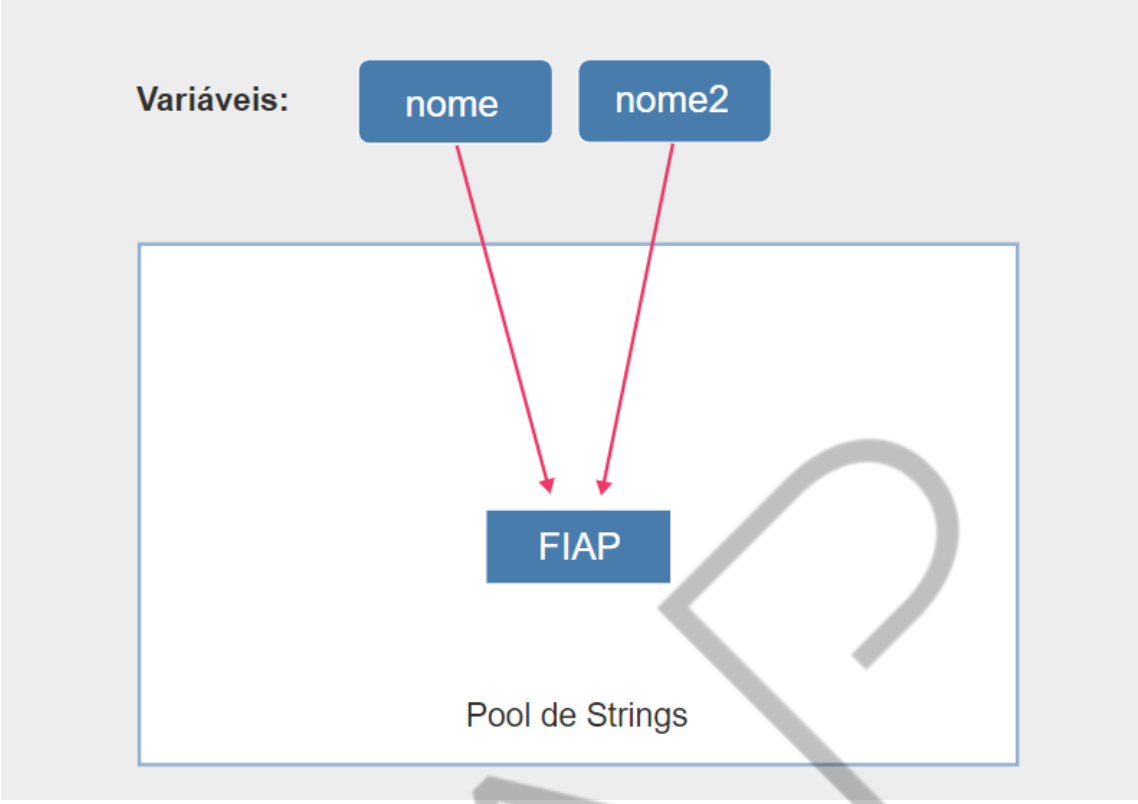


Figura 5.5 – Resultado “As *strings* são iguais”
Fonte: Elaborado pelo autor (2019)

Porém, se uma das duas variáveis for instanciada (new), o operador == não vai funcionar. O exemplo abaixo resulta em “As *strings* são diferentes”:

```
String nome = "FIAP";
String nome2 = new String("FIAP");
if (nome == nome2){
    System.out.println("As Strings são iguais");
}else{
    System.out.println("As Strings são diferentes");
}
```

Por esse motivo é extremamente recomendado sempre utilizar os métodos para realizar a comparação de *strings*, pois eles funcionam independentemente da forma que a variável foi inicializada.

- **Método equals:** compara o conteúdo de duas *strings*, diferenciando os caracteres maiúsculos e minúsculos. Dessa forma, a *string* “Fiap” é diferente de “fiap”. O exemplo abaixo compara o conteúdo das variáveis, assim o valor impresso no console será *“*As *strings* são iguais*”*:

```
String nome = "FIAP";
String nome2 = new String("FIAP");
if (nome.equals(nome2)){
    System.out.println("As Strings são iguais");
}else{
    System.out.println("As Strings são diferentes");
}
```

Como faremos se houver a necessidade em comparar se as *strings* são diferentes? Para isso não precisamos de um método específico. Basta adicionar na comparação o operador de negação (!) e, dessa forma, a comparação será o contrário, ou seja, ao invés de ser igual, será diferente.

Exemplo:

```
if (!nome.equals(nome2)){
    System.out.println("As Strings são diferentes");
}else{
    System.out.println("As Strings são iguais");
}
```

- **Método equalsIgnoreCase:** compara o conteúdo de duas *strings*, mas não diferencia os caracteres maiúsculos e minúsculos. Assim sendo, a *string* “Fiap” é igual à *string* “fiap”.

```
String nome = "fiap";
String nome2 = new String("FIAP");
if (nome.equalsIgnoreCase(nome2)){
    System.out.println("As Strings são iguais");
}else{
    System.out.println("As Strings são diferentes ");
}
```

O resultado do exemplo acima é “As *strings* são iguais”. O operador de negação também pode ser utilizado normalmente com este método, dessa forma, você verifica se as *strings* são diferentes, independentemente das letras maiúsculas e minúsculas.

Podemos verificar se uma *string* começa com uma sequência de caracteres específica. O método **startsWith** recebe a palavra a ser procurada:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
if (facu.startsWith("FIAP")){
    System.out.println("A string começa com FIAP");
}else{
    System.out.println("A string não começa com FIAP");
}
```

Neste exemplo, o resultado é *"*A *string* começa com FIAP*"*. Por padrão, o Java sempre compara as *strings*, diferenciando as letras maiúsculas e minúsculas, somente o método **equalsIgnoreCase**, com o próprio método diz, ignora a diferença entre maiúsculas e minúsculas. Então o método **startsWith** diferencia o *case* das letras.

Podemos verificar se uma *string* começa com uma determinada palavra, também podemos verificar se uma *string* termina com uma sequência de caracteres específica. O método que realiza essa função é o **endsWith,** que também recebe a palavra a ser procurada.

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
if (facu.endsWith("gia")){
    System.out.println("A string termina com gia");
}else{
    System.out.println("A string não termina com gia");
}
```

O resultado é que a *string* termina com a palavra procurada. Esse método também diferencia as letras maiúsculas das minúsculas.

Assim como podemos recuperar o tamanho de um vetor, em uma *string,* podemos recuperar a quantidade de caracteres através do método **length.** Porém, na *string*, o length é um método, assim deve terminar com abre e fecha parênteses:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
int caracteres = facu.length();
System.out.println("A string possui " + caracteres + " caracteres");
```

O resultado da execução será:

```
A string possui 39 caracteres
```

Podemos também recuperar um caractere específico de uma *string* dada a sua posição. Essa funcionalidade é muito parecida com um vetor, no qual podemos recuperar um elemento por meio do seu índice. E assim como o vetor, o primeiro caractere de uma *string* está na posição zero (0).

O método para obter um caractere da *string* é o **charAt.** Este método recebe a posição do caractere que será recuperado:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
char caracter = facu.charAt(1);
System.out.println("O segundo caracter da string é " + caracter);
```

O exemplo acima recupera e imprime no console o segundo caractere da *string* (índice 1). Assim, o resultado será:

```
O segundo caracter da string é I
```

Outro método muito interessante da classe string é o **indexOf.** Esse método permite localizar a primeira ocorrência de um caractere ou palavra em uma *string*. Dessa forma, se for localizado o caractere ou a palavra procurada, o método retorna à posição (índice) da primeira ocorrência da palavra ou caractere. Caso contrário, o valor -1 é retornado, indicando assim que a *string* não possui o valor procurado.

No exemplo abaixo, estamos procurando a primeira ocorrência do caractere ‘a’:

```
String facu = "FIAP - A Melhor Faculdade de Tecnologia"; int
posicao = facu.indexOf('a'); System.out.println("O índice do
caracter 'a' na string é " + posicao);
```

O resultado esperado é:

```
O índice do caracter 'a' na string é 17
```

Lembre-se que o índice da *string* começa no zero e os espaços em branco também são considerados.

Agora vamos realizar uma pequena alteração no exemplo, vamos buscar por um caractere que não existe na *string*:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
int posicao = facu.indexOf('x');
System.out.println("O índice do caracter 'x' na string é " +
posicao);
```

O resultado será:

```
O índice do caracter 'x' na string é -1
```

O resultado -1 indica que o caractere 'x' não está presente na *string*. O método **indexOf** também pode ser utilizado para procurar por uma sequência de caracteres, para isso, basta passar uma palavra ao invés do caractere:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
int posicao = facu.indexOf("Faculdade");
System.out.println("O índice da palavra \"faculdade\" na string é "
+ posicao);
```

A variável posição deve armazenar o valor do índice do início da palavra "faculdade":

```
O índice da palavra "Faculdade" na string é 16
```

Caso a palavra não seja encontrada, o valor -1 também será retornado. Lembre-se de que esse método retorna à primeira ocorrência da palavra ou caractere, dessa forma, no exemplo acima, se a variável *facu* possuir duas palavras "faculdade", o valor retornado será o índice da primeira palavra:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia, Faculdade";
int posicao = facu.indexOf("Faculdade");
System.out.println("O índice da palavra \"faculdade\" na string é "
+ posicao);
```

Resultado:

```
O índice da palavra "Faculdade" na string é 16
```

Outro método da classe string é o **lastIndexOf,** muito parecido com o método **indexOf**, esse método retorna o índice da **última** ocorrência de um caractere ou palavra em uma *string*.

No exemplo abaixo, estamos procurando pela última ocorrência do caractere 'a':

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
int posicao = facu.lastIndexOf('a');
System.out.println("O índice do último caracter \'a\' na string é "
+ posicao);
```

A posição retornada será o index do último caractere a, neste caso a última letra da *string*:

```
O índice do último caracter 'a' na string é 38
```

Para procurar por uma palavra, basta passá-la como parâmetro:

```
String facu = "FIAP - A Melhor Faculdade de Tecnologia"; int
posicao = facu.lastIndexOf("Faculdade"); System.out.println("O
índice da última palavra \"Faculdade\" na string é " + posicao);
```

Neste exemplo, o resultado será o mesmo da busca pela primeira ocorrência de uma palavra:

```
O índice da última palavra "Faculdade" na string é 16
```

Isso aconteceu porque existe somente uma palavra "Faculdade" na *string*. Vamos testar agora com duas palavras iguais:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia, Faculdade";
int posicao = facu.lastIndexOf("Faculdade");
System.out.println("O índice da última palavra \"Faculdade\" na
string é " + posicao);
```

O resultado será o índice da segunda palavra "Faculdade" na *string*:

```
O índice da última palavra "Faculdade" na string é 41
```

É possível criar uma *string* a partir de um trecho de outra *string* utilizando o método **subString.** Este método recebe como parâmetro a posição inicial (inclusive) e a posição final (exclusive) do conjunto de caracteres a serem copiados da *string* original. O caractere da posição inicial será copiado para a nova *string*, já o caractere da última posição não será copiado.

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
String nova = facu.substring(16, 25);
System.out.println("A nova string é: " + nova);
```

O resultado será uma nova string com a palavra “Faculdade”, que está no índice 16 e se finaliza na posição 24:

```
A nova string é: Faculdade
```

Podemos utilizar os métodos visto acima em conjunto ao método **subString**. Por exemplo, podemos utilizar o método **indexOf** para retornar o índice da primeira ocorrência e criar uma nova *string*:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
String nova = facu.substring(facu.indexOf('M'), 25);
System.out.println("A nova string é: " + nova);
```

O resultado da busca do índice do caractere ‘M’ será 9, dessa forma, a nova *string* será iniciada na posição 9 e terminando na posição 25:

```
A nova string é: Melhor Faculdade
```

O método **subString** também aceita apenas a posição inicial do conjunto de caracteres a serem copiados da *string* original. Assim, a nova *string* será criada da posição inicial informada, até o final da *string* original.

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
String nova = facu.substring(16);
System.out.println("A nova string é: " + nova);
```

O resultado será:

```
A nova string é: Faculdade de Tecnologia
```

Também é possível utilizar os métodos em conjunto:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
String nova = facu.substring(facu.lastIndexOf('T'));
System.out.println("A nova string é: " + nova);
```

Dessa forma, o resultado da execução será:

```
A nova string é: Tecnologia
```

Podemos converter os caracteres de uma *string* para maiúsculo ou minúsculo, para isso a classe string possui os métodos **toUpperCase** e **toLowerCase**. Para transformar os caracteres de uma *string* para maiúsculo, podemos utilizar o método **toUpperCase** e para minúsculo o método **toLowerCase.** Lembre-se que uma *string* é imutável, ou seja, ela não pode ser alterada. Dessa forma, quando utilizamos esses métodos, uma nova *string* será criada com a alteração solicitada.

Exemplo:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
String nova = facu.toUpperCase();
System.out.println("A nova string é: " + nova);
```

O resultado será:

```
A nova string é: FIAP - A MELHOR FACULDADE DE TECNOLOGIA
```

Vamos realizar uma pequena alteração no exemplo anterior:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
facu.toUpperCase();
System.out.println("A nova string é: " + facu);
```

Dessa vez, qual será o resultado?

O resultado será a *string* original:

```
A nova string é: FIAP - A Melhor Faculdade de Tecnologia
```

No código acima, o método **toUpperCase** é utilizado para transformar os caracteres em maiúsculo, porém o resultado da transformação não é armazenada em nenhuma variável. Com isso, o resultado será a impressão da *string* original no console. O método **toUpperCase** (ou **toLowerCase**) cria uma nova *string* com a modificação, sem alterar a string original.

Agora, vamos desenvolver um exemplo com o método **toLowerCase:**

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
String nova = facu.toLowerCase();
System.out.println("A nova string é: " + nova);
```

O resultado será a nova *string* armazenada na variável nova com todas as letras em minúsculas:

```
A nova string é: fiap - a melhor faculdade de tecnologia
```

Podemos substituir caracteres ou palavras de uma *string* original. O método ***replace*** recebe como parâmetros o caractere ou palavra a ser substituída e a letra ou palavra para substituir. Esse método também cria uma nova *string* com a alteração:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
String nova = facu.replace('a', 'x');
System.out.println("A nova string é: " + nova);
```

No exemplo acima, estamos substituindo o caractere ‘a’ pelo caractere ‘x’ e estamos atribuindo o resultado dessa nova *string* criada na variável nova. O resultado será:

```
A nova string é: FIAP - A Melhor Fxculdxde de Tecnologix
```

Se imprimirmos o valor da variável facu**,** a *string* original será exibida:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
String nova = facu.replace('a', 'x');
System.out.println("A nova string é: " + nova);
System.out.println("Valor da variável facu: " + facu);
```

O resultado será:

```
A nova string é: FIAP - A Melhor Fxculdxde de Tecnologix Valor da
variável facu: FIAP - A Melhor Faculdade de Tecnologia
```

É possível também substituir uma palavra em uma *string*:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
String nova = facu.replace("Tecnologia", "São Paulo");
System.out.println("A nova string é: " + nova);
System.out.println("Valor da variável facu: " + facu);
```

O resultado da execução do exemplo acima será a nova *string* com as palavras “São Paulo” no lugar da palavra “Tecnologia”:

```
A nova string é: FIAP - A Melhor Faculdade de São Paulo Valor da
variável facu: FIAP - A Melhor Faculdade de Tecnologia
```

O último método da classe string que será estudado será o método **split.** Este é um método muito útil, que separa o valor de uma *string* em várias *strings* separadas por um delimitador, que deve ser informado ao método:

```
String facu = "FIAP – A Melhor Faculdade de Tecnologia";
String[] palavras = facu.split(" ");
for (String p : palavras) {
    System.out.println(p);
}
```

Este exemplo separa a *string* armazenada na variável facu em várias palavras separadas por um espaço. O resultado é armazenado em um vetor de *strings*. Após, utilizamos um laço de repetição para percorrer o vetor e imprimir o valor de seus elementos, o resultado será:

```
FIAP
-
A
Melhor
Faculdade
de
Tecnologia
```

Além de utilizar o espaço como delimitador, podemos utilizar qualquer outro caractere ou palavra:

```
String disciplinas = "LTP;Web;Algoritmos;Banco de Dados";
String[] dis = disciplinas.split(";");
for (String d : dis) {
    System.out.println(d);
}
```

O exemplo acima utilizou o caractere ponto e vírgula (;) como delimitador, dessa forma o resultado da execução será:

```
LTP
Web
Algoritmos
Banco de Dados
```

5.6 *Collections Framework* (COLEÇÕES)

Podemos utilizar um *arrays* para armazenar elementos do mesmo tipo, porém, como vimos, é um processo um pouco trabalhoso, já que não podemos redimensionar o tamanho de um vetor. Portanto, caso seja preciso armazenar mais elementos do que o tamanho do vetor, será necessário criar um novo *array* e repassar todo o conteúdo do *array* antigo para o novo.

Outra dificuldade é encontrar um elemento do vetor pelo seu valor, é possível somente encontrar o elemento pelo seu índice. Por exemplo, se tivermos um vetor de *strings* que armazena os nomes dos alunos, se for preciso procurar por um nome específico no vetor, será necessário percorrer

todo o *array* até encontrar o valor procurado ou então até o termino o vetor, concluindo assim que o elemento não está no *array*.

Quando criamos um *array,* nós determinamos o seu tamanho. Porém, não é possível determinar de uma forma simples a quantidade de posições que foram preenchidas. Temos o conhecimento do tamanho total, contudo, para identificar quantas posições estão vazias ou foram preenchidas, é necessário percorrer todo o *array.* Portanto, pensando novamente no nosso vetor de *strings,* o que é preciso fazer para armazenar um novo nome no *array*? Precisamos procurar por um espaço vazio e para isso é necessário percorrer o *array.* O que acontece se o *array* não possuir espaço vazio? Será necessário criar um novo *array* e copiar os dados antigos para ele.

Por esses e outros motivos é que a plataforma Java tem um conjunto de classes e interfaces conhecido como **Collections Framework,** que representam estruturas de dados avançadas.

Collections Framework ou Coleções são estruturas de dados utilizadas para armazenar e organizar objetos de maneira eficiente e prática. Podem ser utilizadas para representar estruturas como vetores, listas, pilhas, filas, mapas, conjuntos e outras estruturas de dados.

Coleções são muito comuns nas aplicações Java, principalmente para o acesso ao banco de dados, principalmente no resultado de buscas. Assim, podemos armazenar os clientes, livros, endereços em nossa aplicação.

As coleções são definidas por meio de interfaces. As interfaces determinam o que a estrutura deve fornecer de funcionalidades, ou seja, fornecem um contrato para que a classe concreta as implemente. Na figura a seguir é apresentada a estrutura de interfaces e classes das coleções e mapas do Java:

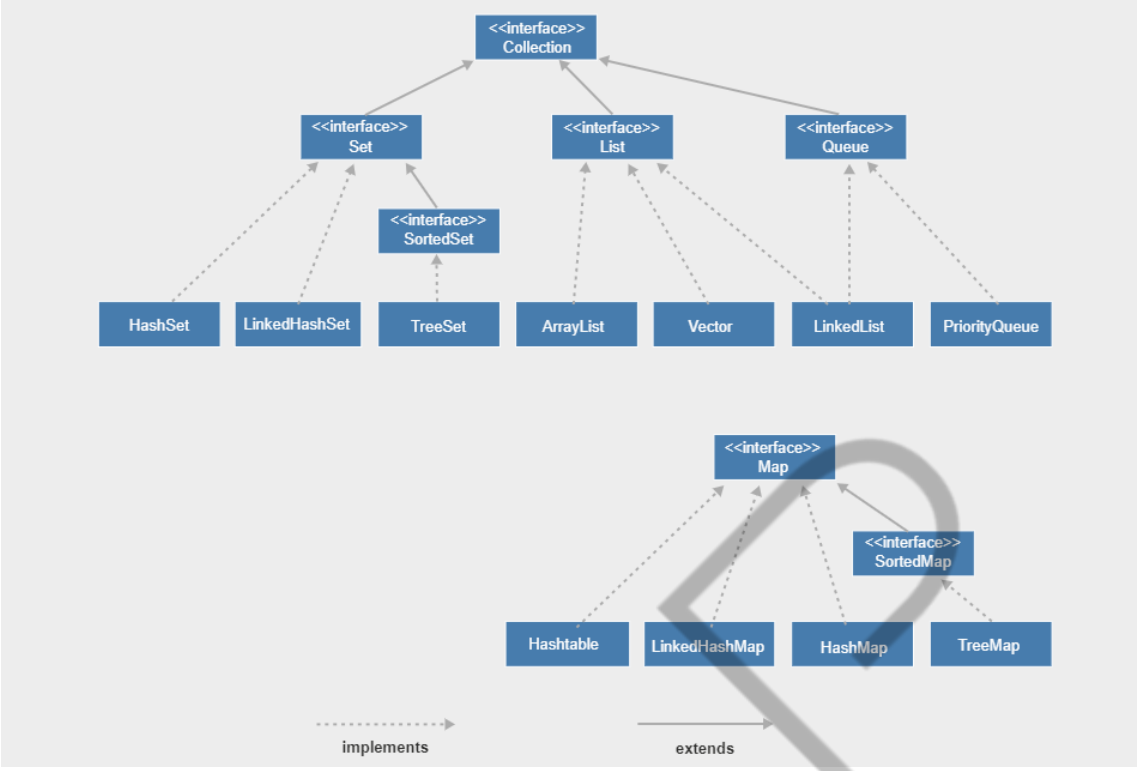


Figura 5.6 – Estrutura de coleções e mapas
Fonte: Elaborado pelo autor (2019)

No Java, as coleções podem ser classificadas em duas categorias: as que implementam a interface **Collection** e as que implementam a interface **Map.**

As principais subinterfaces de **Collection** são:

- **List:** representa uma lista de objetos, a implementação mais utilizada é o **ArrayList.**
- **Set:** representa um conjunto de objetos únicos, e os objetos não se repetem; a implementação mais utilizada é a **HashSet.**

A Interface **Map** representa uma tabela Hash, que armazena valores compostos por [chave, valor]. Ou seja, para cada valor armazenado nesse mapa podemos definir uma chave para facilitar a recuperação do valor. A principal subinterface é:

- **SortedMap:** representa um mapa ordenado, a implementação mais utilizada é o **HashMap.**

A interface **Collection** que é base para todas as coleções, exceto Mapas, define um conjunto de métodos que são comuns a todas outras estruturas que estão abaixo dela: list, set e queue.

Os principais métodos da interface Collection estão apresentados no quadro a seguir:

Método	Descrição
add	Adiciona um objeto a coleção
clear	Remove todos objetos da coleção
contains	Verifica se a coleção contém o objeto determinado
isEmpty	Verifica se a coleção está vazia
remove	Remove um objeto da coleção
size	Retorna O quantidade de objetos na coleção
toArray	Retorna uma array contendo os elementos da coleção

PRINCIPAIS MÉTODOS DA INTERFACE COLLECTION

Quadro 5.2 – Principais métodos da interface Collection
Fonte: FIAP (2019)

5.6.1 List

A interface **List** representa uma sequência de elementos ordenados e podem conter elementos repetidos. Dessa forma, os elementos de uma lista estão dispostos pela ordem de inserção.

Para criar uma lista, não precisamos passar o tamanho dela, como temos que fazer no *array*. Isso quer dizer que a lista se adequa quando inserimos um elemento, possibilitando adicionar ou remover quantos elementos forem necessários. Outra grande vantagem é que podemos manipular a lista, ordenando-a ou buscando um elemento pelo seu valor.

A interface **List** especifica as funcionalidades que as classes devem implementar para uma lista. Há diversas implementações disponíveis, sendo a **ArrayList** a mais utilizada**.**

Principais métodos de uma lista:

Método	Descrição
Add	Adiciona um objeto numa determinada posição
get	Retorna o objeto localizado numa determinada posição
Remove	Remove um objeto localizado numa determinada posição
set	Coloca um objeto numa determinada posição (substitui objetos)
indexOf	Retorna a posição de um objeto na lista
lastIndexOf	Retorna a última posição de um objeto na lista
subList	Retorna parte de uma lista

Quadro 5.3 – Principais métodos de uma lista
Fonte: FIAP (2019)

A **ArrayList** armazena seus elementos em um *array* interno para gerar uma lista. Essa lista cresce ou diminui dinamicamente no momento que um elemento é inserido ou excluido da lista. Apesar do nome ArrayList, não é um *array*, como visto anteriormente. ArrayList é uma implementação da interface List da API de Collections do Java, essa classe somente utiliza um *array* (também já visto neste capítulo) para armazenar os valores, porém, não podemos acessar diretamente esse *array*, pois é um atributo encapsulado.

Para criar uma ArrayList, basta chamar o seu construtor:

```
ArrayList lista = new ArrayList();
lista.add("LTP");
lista.add("Web");
lista.add("Algoritmos");
```

No exemplo acima, declaramos uma ArrayList chamada lista e utilizamos o método **add,** passando como parâmetro um objeto, para ser inserido na lista, esse objeto será adicionado ao final da lista.

Outro método utilizado para inserir um elemento na lista é o **set.** Esse método adiciona um elemento em determinada posição da lista, substituindo o valor existente nele, ou seja, quando quisermos inserir algo na primeira posição e essa já estiver ocupada, utilizamos o método set determinando a posição desejada, que para o caso seria 0, já o objeto que estava nessa posição é excluído da ArrayList:

```
ArrayList lista = new ArrayList();
lista.add("LTP");
lista.add("Web");
lista.set(1,"Algoritmos");
```

No exemplo acima, no final da execução, a lista possuirá somente dois elementos, na posição 0 o valor “LTP” e na posição 1 o valor “Algoritmos”.

Para remover um elemento, utilizamos o método **remove**, passando como parâmetro a posição do elemento a ser removido:

```
ArrayList lista = new ArrayList();
lista.add("LTP");
lista.add("Web");
lista.remove(1);
```

Neste exemplo, foram adicionados dois elementos na lista e depois removido o elemento da posição 1, ou seja, o segundo elemento. Dessa forma, somente o valor “LTP” está armazenado na lista.

O método **get** recupera um elemento dado a sua posição:

```
ArrayList lista = new ArrayList();
lista.add("LTP");
lista.add("Web");

System.out.println(lista.get(1));
```

O valor que será impresso no console será “Web”.

Podemos percorrer uma lista como fizemos com *arrays*. No exemplo abaixo, vamos percorrer a lista e imprimir o valor de cada posição dele:

```
ArrayList lista = new ArrayList();
lista.add("LTP");
lista.add("Web");
lista.add("Algoritmo");

for (int i = 0; i < lista.size(); i++) {
    System.out.println(lista.get(i));
}
```

O método **size()** retorna a quantidade de elementos na coleção. O resultado da execução será:

```
LTP
Web
Algoritmo
```

Para procurar por um valor em uma ArrayList, podemos utilizar o método **indexOf** ou **lastIndexOf,** que são muito parecidos com os métodos da classe string, visto anteriormente. Como uma lista pode conter elementos duplicados, o primeiro método **indexOf** retorna ao índice da primeira ocorrência encontrada na lista:

```
ArrayList lista = new ArrayList();
lista.add("LTP");
lista.add("Web");
lista.add("Algoritmo");

int indice = lista.indexOf("Web");

System.out.println("O valor \"Web\" está na posição: " + indice);
```

O resultado da execução será:

```
O valor "Web" está na posição: 1
```

O método **lastIndexOf()** funciona de forma parecida com o método da classe string, recuperando o índice da última ocorrência na lista. Os dois métodos retornam o valor **–1,** caso o elemento procurado não esteja presente na lista.

5.6.2 Set

A interface **Set** define uma coleção que não pode conter valores duplicados. Corresponde à abstração de um conjunto que funciona de forma análoga aos conjuntos da matemática. Outro ponto importante, é que nem sempre a ordem de inserção dos elementos será a dos elementos dispostos na coleção, isso pode variar de implementação para implementação. A interface contém somente os métodos herdados da interface **Collection:**

Método	Descrição
add	Adiciona um objeto no Set
clear	Remove todos objetos do Set
contains	Verifica se o Set possui um objeto determinado
isEmpty	Verifica se o Set está vazio
remove	Remove um Objeto do Set
size	Retorna o quantidade de objetos no Set
toArray	Retorna uma array contendo os objetos do Set

Quadro 5.4 – Métodos herdados da interface Collection
Fonte: FIAP (2019)

Uma das principais implementações de **Set** é a classe **HashSet,** que armazena seus elementos em uma tabela hash. É uma implementação bastante simples e eficiente, como mostra o exemplo a seguir:

```
HashSet cursos = new HashSet<>();

cursos.add("Java");
cursos.add(".NET");
cursos.add("Android");

cursos.add("Java"); //Repetido!

//Imprime todos os elementos
System.out.println(cursos);
```

Como se vê, o segundo valor “Java” não será adicionado ao HashSet. O resultado será:

```
[Java, .NET, Android]
```

A grande vantagem do **Set** é a performance nas operações de busca (método contais), em relação a **List.**

5.6.3 Map

Podemos armazenar informações em mapas, eles são muito úteis quando precisamos recuperar de forma rápida as informações do objeto, para

isso é preciso passar uma chave. Por exemplo, podemos armazenar o objeto **Aluno** em um mapa e utilizar o **RM** como chave. Dessa forma, é possível informar o RM do aluno para recuperar o objeto que tem todas as informações do aluno.

Podemos utilizar uma lista para isso? Claro que sim, o problema é que seria necessário percorrer todos os elementos da lista para encontrar o aluno correto, deste modo, a performance seria comprometida, mesmo para pequenas listas.

Um mapa é composto por um par de **chave** e **valor**. As chaves não podem conter valores iguais, porém o valor sim. A principal implementação de **Map** é a classe **HashMap.**

Principais definições da interface **Map** são**:**

Método	Descrição
clear	Remove todos os mapeamentos
containsKey	Verifica se uma chave já está presente no mapeamento
containsValue	Verifica se um valor já está presente no mapeamento
get	Retorna o valor associado a uma chave determinada
isEmpty	Verifica se o mapeamento está vazio
kaySet	Retorna um Set contendo as chaves
put	Adiciona um mapeamento
remove	Remove um mapeamento
size	Retorna o número de mapeamentos
values	Retorna o número de mapeamentos

Quadro 5.5 – Definições da interface Map
Fonte: FIAP (2019)

O método **put** é utilizado para adicionar um elemento ao Mapa**.** Esse método recebe a chave e o valor a ser inserido:

```
HashMap mapa = new HashMap();

mapa.put("RM1234", "Thiago");
mapa.put("RM4321", "João");

System.out.println(mapa);
```

Para adicionar os elementos no mapa, foram utilizados os pares rm e nome do aluno para a chave e valor, respectivamente.

O resultado da execução será:

```
{RM4321=João, RM1234=Thiago}
```

Para recuperar um elemento do mapa, basta utilizar o método **get** passando a chave:

```
System.out.println(mapa.get("RM1234"));
```

O resultado será a impressão do valor “Thiago” no console.

As operações de inserir (put) e buscar (get) são as mais utilizadas em mapas. Também existem métodos para remover um elemento do mapa. Já para excluir, utilizamos o método **remove**, que recebe como parâmetro a chave do elemento a ser removido:

```
HashMap mapa = new HashMap();

mapa.put("RM1234", "Thiago");
mapa.put("RM4321", "João");

mapa.remove("RM1234"); //remove um elemento

System.out.println(mapa.get("RM1234"));
```

O elemento com a chave “RM1234” foi removido do mapa, assim, quando tentarmos recuperar o valor da chave removida, o valor retornado será *null* (vazio).

5.6.4 Generics

Até o momento nós armazenamos qualquer tipo de objeto nas coleções. Com isso, podemos misturar os objetos dentro de uma mesma coleção.

```
ArrayList lista = new ArrayList();

Aluno aluno = new Aluno();

lista.add("FIAP"); //String
lista.add(2); //Integer
lista.add(aluno); //Aluno
```

E para recuperar o elemento da lista? O método **get** retorna um objeto e, depois, é preciso realizar um cast. Mas com uma coleção com objetos diferentes, será complicado realizá-lo.

A partir do Java 5, podemos utilizar o recurso de Generics para restringir os tipos de dados aceitos por referência genérica. Dessa forma, somente o tipo determinado no Generic será permitido inserir na lista, e não qualquer objeto.

O Generic permite a verificação do tipo em tempo de compilação e deixa o código mais limpo, pois não é necessário realizar um cast.

Sintaxe:

```
ArrayList<Tipo> lista = new ArrayList<Tipo>();
```

Ao lado do ArrayList temos um < > e dentro dele, determinamos o tipo que a lista poderá armazenar. A figura a seguir apresenta os detalhes da declaração de uma ArrayList utilizando Generics:

Figura 5.7 – Detalhes da declaração de uma ArrayList utilizando Generics
Fonte: Elaborado pelo autor (2019)

Essa ArrayList poderá armazenar somente objetos do tipo produto.

Sem utilizar o Generic, seria necessário realizar o cast e, possivelmente, ocorreria uma exceção se o tipo não fosse compatível com o cast, como indicado a seguir:

```
ArrayList listaCliente = newArrayList();

Cliente cliente01 = newCliente();
listaCliente.add(cliente01); //[0]

Cliente cliente02 = (Cliente)listaCliente.get(0);

Fornecedor fornecedor = newFornecedor();
listaCliente.add(fornecedor); //[1]

Cliente cliente03 = (Cliente)listaCliente.get(1); //exception
```

Exceção:
java. lang . ClassCastException

Figura 5.8 – Exceção quando o tipo não é compatível com o cast
Fonte: Elaborado pelo autor (2019)

Agora com o Generics, o código não permite a inserção de um elemento que não corresponda ao tipo de objeto da lista:

```
ArrayList<Cliente> listaCliente = newArrayList<Cliente>();

Cliente cliente01 = newCliente();
listaCliente.add(cliente01); //[0]

Cliente cliente02 = listaCliente.get(0); //sem cast

Fornecedor fornecedor = newFornecedor();
listaCliente.add(fornecedor); //não compila!
```

Não compila:
Esta lista só aceita objetos do tipo Cliente

Figura 5.9 – Com Generics, código não permite inserção de elemento que não corresponde ao tipo de objeto da lista.
Fonte: Elaborado pelo autor (2019)

Além disso, não será preciso realizar o cast para obter o objeto de volta na lista, pois ela pode armazenar somente um tipo de objeto:

```
ArrayList<Cliente> listaCliente = new ArrayList<Cliente>();

Cliente cliente1 = new Cliente();
cliente1.setNome("Thiago");
Cliente cliente2 = new Cliente();
cliente2.setNome("João");

listaCliente.add(cliente1);
listaCliente.add(cliente2);

for (int i = 0; i < listaCliente.size(); i++) {
    Cliente cli = listaCliente.get(i); //Não é necessário o cast
    System.out.println(cli.getNome());
}
```

Podemos utilizar o operador **for-each** para percorrer a lista:

```
for (Cliente cliente : listaCliente) {
    System.out.println(cliente.getNome());
}
```

Toda a API de Collections permite a utilização de Generics por permitir maior segurança na manipulação dos tipos e não há necessidade de cast.

Lembre-se que o Generic permite definir um tipo de Objeto, os tipos primitivos não são permitidos:

```
ArrayList<int> ou HashMap<int,double> não são válidos.
```

Assim, podemos determinar o tipo que será armazenado no Set e Map também:

```
HashSet<Cliente> conjunto = new HashSet<Cliente>();

HashMap<String, Cliente> mapa = new HashMap<String, Cliente>();
```

Portanto, sempre é recomendado utilizar as coleções com generics. No exemplo acima, podemos somente armazenar no **Set** os objetos do tipo Cliente. Já no **Map**, a chave será uma String e o valor um Cliente.

REFERÊNCIAS

BARNES, David J. **Programação Orientada a Objetos com Java**: Uma introdução Prática Utilizando Blue J. São Paulo: Pearson, 2004.

CADENHEAD, Rogers; LEMAY, Laura. **Aprenda em 21 dias Java 2 Professional Reference.** 5. ed. Rio de Janeiro: Elsevier, 2003.

DEITEL, Paul; DEITEL, Harvey. **Java Como Programar**. 8. ed. São Paulo: Pearson, 2010.

HORSTMANN, Cay; CORNELL, Gary. **Core Java**. Fundamentos. v.1. 8. ed. São Paulo: Pearson 2009.

SIERRA, Kathy; BATES, Bert. **Use a cabeça! Java.** Rio de Janeiro: Alta Books, 2010.