Paradigma de programação

Para começarmos a falar sobre paradigmas de programação é importante entendermos antes de tudo que quando falamos de paradigmas de programação, não estamos falando de **linguagens** de programação, são coisas distintas.

Então o que são paradigmas de programação?

Podemos dizer **serem um estilo de programação ou um tipo de estruturação ao qual a linguagem deverá respeitar**. Da mesma maneira que existem diversas linguagens de programação, temos também diversas formas de programar. Se trata da **forma como você soluciona problemas utilizando determinada linguagem**.

Vamos a um exemplo:

Imagine que você precisa realizar uma determinada entrega em um local x. Existem diversas maneiras de você realizar essa tarefa: você pode ir até o local usando seu carro, bicicleta, carro de aplicativo etc. A maneira como irá realizar essa tarefa deve ser escolhida com base na que seja melhor para você considerando fatores como tempo e custo, que dependem de algumas análises, por exemplo: a distância entre o seu local e o local de entrega.

Em programação, também podemos recorrer a diferentes maneiras para alcançar um mesmo objetivo.

A essas maneiras damos o nome de paradigmas.

Alguns paradigmas de programação existentes

* Paradigma Imperativo

Neste paradigma, também chamado de *procedural*, o programador escreve uma lista de instruções, um passo-a-passo e sequência do que deverá ser executado pela máquina.

* Paradigma Funcional

A base deste paradigma é a execução de uma série de *funções*, ou seja, uma série de blocos de código.

* Paradigma Lógico

Este paradigma utiliza *avaliações lógico-matemáticas* como padrões de entrada e saída.

* Paradigma Orientado a Objetos

A programação orientada a objetos surgiu com o objetivo de aproximar a programação do mundo real, trazendo os objetos como algo genérico que pode *representar algo do nosso mundo real* e categorizados em tipos.

Ao compreendermos paradigmas de programação podemos nos orientar e identificar qual paradigma de programação se adapta mais e irá nos trazer melhor resultado para a solução que queremos desenvolver.

Materiais complementares

* <https://cs.lmu.edu/~ray/notes/paradigms/>

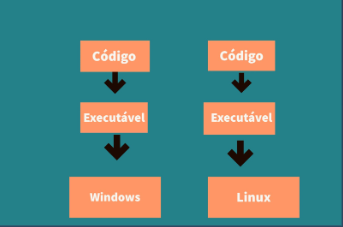
Referências

* <https://blog.geekhunter.com.br/quais-sao-os-paradigmas-de-programacao/>
* <https://guia.dev/pt/pillars/languages-and-tools/programming-paradigms.html>

# Introdução ao Java

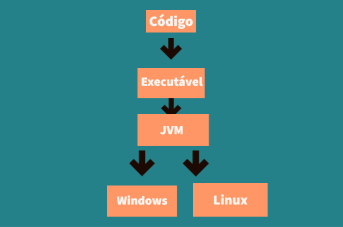
Java é uma linguagem de programação orientada a objetos, criada pela Sun Microsystems em 1995 que, posteriormente, foi adquirida pela Oracle. Apesar disso, a linguagem Java (que também pode ser referida como uma plataforma) é mantida pela JCP (Java Community Press) que organiza as discussões e mudanças através de JSR (Java Specification Requests).

Com o sucesso da internet, também surgiram problemas. Um deles é que, com diversos sistemas operacionais surgindo, um código era escrito por um determinado sistema operacional, gerava um executável, como o ".exe" (no Windows) e esse código só funcionava naquele determinado sistema operacional.



**Figura 1** – Arquitetura\_execução (Fonte da imagem:da autora)

Com isso, foi decidido resgatar uma ideia que não foi bem-sucedida para problemas anteriores e aplicá-la no novo problema. Um código escrito em uma única linguagem, o Java, iria gerar um executável, o bytecode Java. Esse executável seria lido e entendido por uma máquina virtual (JVM) e convertido para ser executado no sistema operacional destino.



**Figura 2** – resolução (Fonte da imagem:da autora)

O principal diferencial do Java quando foi lançado era o seu suporte multiplataforma, propagado sob o slogan "Write once, run everywhere". Comparada às linguagens da época, distribuídas na forma de código-objeto, Java trazia o diferencial de que seu código era compilado para uma linguagem intermediária (o bytecode), sendo este interpretado por uma **Java Virtual Machine (JVM).**Esse ambiente virtual, sim, sendo dependente de sistema operacional e arquitetura de processadores.

#Diagrama

Descrição gerada automaticamente

**Figura 3** – Uma visão geral do processo de desenvolvimento de software (Fonte da imagem: [Docs oracle](https://docs.oracle.com/))

## **JDK e JRE**

* **Java Development Kit - JDK**é basicamente, um conjunto de utilitários, um pacote de software que nos entrega todo o ambiente necessário para que seja possível desenvolver aplicações em Java como bibliotecas, JVM e o compilador.
* **Java Runtime Environment**é uma camada de software, que é executado sobre um sistema operacional, e nos fornece os recursos necessários (bibliotecas de classe e a JVM) para ter um ambiente que seja possível executar programas em Java.

## **Configuração de ambiente**

Para fazer a configuração do seu ambiente local Java, você precisa:

1. Fazer o download de uma JDK no link a seguir, escolher a versão do Java que você quer instalar e a versão compatível com seu sistema operacional. [Download JDK](https://www.oracle.com/java/technologies/downloads/)
2. Criar a variável de ambiente $JAVA\_HOME e adicionar a pasta $JAVA\_HOME/bin ao $Path. Você pode verificar como criar essa variável de ambiente de acordo com seu sistema operacional no link a seguir. (<https://www.java.com/pt-BR/download/help/path_pt-br.html>)
3. Adquirir/instalar uma IDE. Recomendamos o uso do IntelliJ Idea Community (essa é a versão gratuita, enquanto a Ultimate é a versão paga). [Download IntelliJ](https://www.jetbrains.com/pt-br/idea/download/)

Agora que entendemos um pouco mais da história do Java, suas ferramentas e como podemos obter um ambiente para a criação de nossas aplicações, podemos começar a nos aprofundar na linguagem nos próximos tópicos.

### Materiais complementares

* <https://www.oracle.com/java/technologies/javase-documentation.html>

### Referências

* <https://www.devmedia.com.br/java-historia-e-principais-conceitos/25178>
* <https://www.oracle.com/java/technologies/javase-documentation.html>

# Operadores

Assim como temos os operadores matemáticos da aritmética, álgebra, trigonometria, cálculo etc. O Java define símbolos especiais para realizar operações com nossas variáveis também. Esses operadores podem tomar um, dois ou três termos e agir sobre eles.

* Aritméticos: soma (+), subtração (-), multiplicação (\*) e divisão (/);
* Lógicos: negação (!), E (&&), OU (||) // duplo pipe
* Relacionais: maior que (>), menor que (<), igual (==), diferente (!=)
* Ternário: ? :

Existem algumas variações desses operadores que encurtam as expressões que escrevemos. Por exemplo, ao invés de:

int contador = 0;

contador = contador + 1;

Podemos usar o operador unário ++ na forma pós-fixada contador++ ou até mesmo, pré-fixada ++contador. A diferença é que, ao usar a forma pós-fixada, o valor atual de contador é informado e, em seguida, seu valor é adicionado de 1. Ao contrário, na forma pré-fixada, primeiro é realizado o acréscimo e, depois, o novo valor é retornado.

Um operador aritmético especial, e que não foi mencionado, é o mod %. Este operador informa o "resto da divisão", por exemplo:

int resultado = 3 % 2; // resultado = 1

O operador mod é, frequentemente, usado para verificar se um número é par ou ímpar. (Esse exemplo fica como exercício).

## **Precedência**

Assim como na Matemática, costumamos usar vários operadores em uma mesma expressão. Para que o resultado não nos surpreenda, precisamos conhecer bem a precedência dos operadores. Ou seja, em ordem de prioridade dos operadores e como serão executados.

| **Operators** | **Precedence** |
| --- | --- |
| postfix | exp++ exp-- |
| unary | ++exp --exp +exp -exp ~ ! |
| multiplicative | \* / & |
| additive | + - |
| relational | < > <= >= |
| equality | == != |
| logical AND | && |
| logical OR | || |
| ternary | ? : |
| assignment | = += -= \*= /= %= |

### Materiais complementares

* <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>

### Referências

* <https://docs.oracle.com/javase/tutorial/java/index.html>

# Saídas

É comum querermos apresentar alguma mensagem e/ou resultado para usuário durante a criação de nossos programas, podemos realizar isso através do terminal do computador.

Como já vimos até aqui, podemos realizar isso utilizando as saídas em Java que podem ser feitas através da instrução:

System.out.println(expression);

É desta forma que escrevemos nosso primeiro programa "Hello World":

System.out.println("Hello World");// expected result > Hello World

Neste exemplo, temos que a expressão a ser escrita é uma simples string. Podemos variar este mesmo comportamento declarando essa string em uma variável e informando a variável como expressão:

String greeting = "Hello World";

System.out.println(greeting);// expected result > Hello World

Ainda podemos incrementar mais e usar os operadores que aprendemos para combinar (concatenar) strings. Vamos declarar uma variável com o nome de alguém e emitir uma saudação personalizada:

String name = "John Doe";

System.out.println("Hello, " + name);// expected result > Hello, John Doe

Mas se ainda quisermos mais liberdade para formatar nossa saída, podemos optar pelo método printf. Sua sintaxe é:

System.out.printf(format, arguments);

Em format nós escrevemos a expressão que será escrita em console usando a sintaxe Format String Syntax. Essa sintaxe permite o uso de marcadores de argumentos, ou seja, cada marcador define uma variável que será substituída pelo valor contido nos arguments. O segundo parâmetro, arguments, contém uma sequência de valores, separados por vírgulas, correspondente a cada marcador definido no primeiro parâmetro (format).

Vamos analisar o seguinte exemplo:

String name = "John Doe";

System.out.printf("Hello, %s.", name);// expected result > Hello, John Doe.

Apesar de ser muito semelhante ao exemplo anterior, perceba que a sintaxe é um pouco mais simples. Usando a sintaxe Format String Syntax definimos o marcador %s, o qual define um valor do tipo String. Existe um marcador específico para cada tipo de dados:

* s – formats strings
* d – formats decimal integers
* f – formats the floating-point numbers
* t – formats date/time values

# Entradas

Agora que já aprendemos como escrever valores em console, vamos aprender um pouco sobre como ler esses valores. Isso nos levará a outro nível de poder de programação com a linguagem Java.

Até aqui, todos os valores de variáveis com os quais trabalhamos foram definidos por nós no próprio código-fonte. Isso "amarrou" nossos programas a sempre funcionarem da mesma forma a cada execução. Agora, no entanto, poderemos ler valores inseridos pelos usuários e deixar nosso programa mais dinâmico.

## **String[] args**

A forma mais simples de ler entradas do usuário do programa é através do array args contido na assinatura do método main. Vamos relembrar a sintaxe deste método:

public static void main(String[] args)

O array args é preenchido com os argumentos informados ao nosso programa durante a inicialização dele. Isto é, quando vamos executar nosso programa, podemos definir valores, separados por espaços, como argumentos.

Por exemplo, o programa a seguir deve receber o nome do usuário como argumento e exibir uma saudação personalizada - veja que já estamos aplicando os conceitos de formatação vistos anteriormente:

public class Aplicacao {

public static void main(String[] args) {

System.out.printf("Hello, %s.", args[0]);

}

}

E, então, executamos o programa com o comando em terminal:

java Aplicacao "John Doe"

A saída esperada é Hello, John Doe.

Observe, porém, que o array args é do tipo String. Ou seja, caso você deseje usá-lo para receber um número ou algum dos demais tipos válidos, você precisa fazer a conversão para o tipo correto. Por exemplo:

public class Aplicacao {

public static void main(String[] args) {

int numero = Integer.parseInt(args[0]);

System.out.printf("Você informou o número %d.", numero);

}

}

Para a execução:

java Aplicacao 2

A saída esperada é Você informou o número 2.

## **java.util.Scanner**

A abordagem usando o array args serve apenas para cenários simples, onde todos os argumentos do programa serão informados logo em sua inicialização, mas não nos permite realizar algo interativo, onde vamos recebendo "inputs" passo-a-passo do usuário ou definindo o comportamento da aplicação de forma dinâmica.

Por exemplo, baseando-se no exercício anterior, onde convertemos um peso de quilogramas para libras, agora vamos converter o peso de 2 pessoas:

public class Aplicacao {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Informe o peso da primeira pessoa em kg");

double peso1 = sc.nextDouble();

System.out.println("Informe o peso da segunda pessoa em kg");

double peso2 = sc.nextDouble();

final float FATOR\_CONVERSAO\_KG\_LB = 2.20462f;

System.out.printf("O peso da primeira pessoa em libras é %flb.%n", (peso1 \* FATOR\_CONVERSAO\_KG\_LB));

System.out.printf("O peso da segunda pessoa em libras é %flb.%n", (peso2 \* FATOR\_CONVERSAO\_KG\_LB));

}

}

A primeira linha do nosso programa (dentro do método main) é a declaração de uma variável do tipo Scanner. Através desse tipo especial vamos ler os valores do console, sempre de acordo com o tipo de dado esperado. Veja que, como esperamos um valor do tipo double, usamos a funcionalidade nextDouble.

### Materiais complementares

* <https://www.baeldung.com/java-printstream-printf>
* <https://docs.oracle.com/javase/tutorial/java/data/numberformat.html>

### Referências

* <https://docs.oracle.com/javase/tutorial/java/index.html>
* <https://www.baeldung.com/java-printstream-printf>

## **Funções**

Em Java, chamamos/damos o nome de **métodos** para funções, que são blocos de códigos responsáveis por executar ações ou procedimentos em seu projeto.

Os métodos em Java devem sempre ser definidos em uma classe e é composto por modificador de acesso, tipo do retorno, nome do método e seus argumentos, veja a seguir a sintaxe de um método.

[modificador] tipo\_retorno nome ([argumentos]) {

//conteudo do método

}

1. Modificadores de acesso: são utilizados para definir o tipo de acesso que aquele método terá. Podem ser:
   * public: Pode ser acessado por qualquer outra classe do projeto
   * protected: Podem ser acessados pelos métodos da própria classe, pelas classes do mesmo pacote e pelas classes derivadas.
   * private: Só podem ser acessados pelos métodos da própria classe.

Ainda existe uma quarta opção que é quando omitimos o modificador. Nesse caso, chamamos de visibilidade "default", a qual permite o acesso pelos métodos da própria classe e do mesmo pacote.

Exemplos:

public int somaDoisNumeros(int numero1, int numero2){

return numero1 + numero2;

}

private int subtrairDoisNumeros(int numero1, int numero2){

return numero1 - numero2;

}

protected int dividirDoisNumeros(int numero1, int numero2){

return numero1 / numero2;

}

1. Nome do método: é o nome que damos para aquele bloco de código, o nome que damos a um método deve remeter a aquilo que o método, realiza, se você cria um método que tem como objetivo somar dois números, você pode colocar o nome de somaDoisNumeros.
2. Tipo do retorno: é o tipo que será retornado ao final da execução daquele método para quem o chamar, como int, string, utilizamos void, quando o retorno será vazio.
3. Argumentos: também chamados de parâmetros, são valores que devem ser passados ao método quando chamados, um método pode ter nenhum, um ou mais parâmetros. O nome do método acrescido de seus parâmetros é denominado **assinatura do método**.

### Quando utilizar métodos

Podemos utilizar métodos quando queremos isolar alguma execução ou funcionalidade em um projeto, principalmente se for algo que precisará ser executado muitas vezes e/ou em diferentes partes do código, com isso não escrevemos código repetido e podemos apenas chamar o método criado onde queremos executar essa ação.

Segue exemplo de um método

public int somaDoisNumeros(int numero1, int numero2){

return numero1 + numero2;

}

### Método main

Quando um programa Java é executado a JVM tenta sempre chamar o método main da classe que é onde geralmente iniciamos o nosso projeto. Quando declaramos a main como static é permitido que a JVM chame a classe main sem precisar criar uma instância da mesma, dessa forma, ela é conhecida como classe principal do projeto.

public static void main(String[] args) {

//codigo

}

### Materiais complementares

* <https://www.devmedia.com.br/trabalhando-com-metodos-em-java/25917>

### Referências

* <https://docs.oracle.com/javase/tutorial/java/index.html>
* <https://www.devmedia.com.br/trabalhando-com-metodos-em-java/25917#:~:text=Os%20m%C3%A9todos%20s%C3%A3o%20conhecidos%20como,a%20constru%C3%A7%C3%A3o%20de%20novos%20sistemas>.

Separação de Responsabilidade

Quando falamos em dividir o código em funções ou métodos (nome mais apropriado dentro da linguagem Java), é comum vir a dúvida: *quando eu sei que devo separar o código num método próprio?* É sobre isso que vamos discutir nesse tópico.

*O livro Código Limpo é um ótimo referencial para todo programador do mundo da Programação Orientada a Objetos. Especialmente, os desenvolvedores Java! É verdade que ainda não falamos sobre "orientação a objetos", mas sobre essa discussão de separação de métodos o autor dedica o capítulo 3 inteiro para isso!*

Quando um código está difícil de ser compreendido ou alterado, é um bom sinal que ele está longo demais, assumindo **muitas responsabilidades**. Considere o seguinte código:

public class Aplicacao {

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

System.out.println("Nome:");

String nome = input.nextLine();

System.out.println("Data de nascimento:");

String dataNascimento = input.nextLine();

System.out.println("Endereço (logradouro, número, bairro, cidade, estado):");

String endereco = input.nextLine();

System.out.println("Complemento:");

String complemento = input.nextLine();

System.out.println("Profissão:");

String profissao = input.nextLine();

System.out.println("E-mail:");

String email = input.nextLine();

System.out.println("Telefone:");

String telefone = input.nextLine();

System.out.println("Pessoa { "

+ "nome = "+ nome

+ "Data de nascimento: = "+ dataNascimento

+ "Endereço = "+ endereco

+ "Complemento = "+ complemento

+ "Profissão = "+ profissao

+ "E-mail = "+ email

+ "Telefone = "+ telefone

+" }");

}

}

Por mais que seja um código simples, não é tão fácil de entender. E se surgir a necessidade de adicionar mais dados, vamos sempre ter que voltar e alterar algo que já estava funcionando bem.

Pequenas

*"A primeira regra para funções é que elas devem ser pequenas. A segunda é que devem ser ainda menores." (Robert C. Martin)*

Não existe um estudo que comprove a eficiência de métodos pequenos, que eles sejam mais rápidos ou melhores, ou que gerem menos erros. O fato é que ao restringir ao máximo sua responsabilidade, o código fica mais coeso e, por consequência, mais legível.

Faça apenas uma coisa

Sempre que você escrever um código, imagine-se explicando-o para um colega. No livro "O programador pragmático" (David Thomas, Andrew Hunt), os autores descrevem a cena de um desenvolvedor explicando o comportamento do código para um pato de borracha em sua mesa. Esse momento de "olhar com outros olhos", ou de um ponto de vista diferente, é fundamental em todo processo criativo. E na programação não é diferente.

Se você percebeu - durante seu exercício de explicar o código para um colega - que você está detalhando demais, ou que haveriam muitos "por quê" do seu colega, é bem provável que seu código está fazendo **muitas coisas**. Por exemplo, considere os seguintes passos:

1. Pegue seu telefone
2. Pegue sua carteira
3. Pegue sua chave
4. Abra a porta de casa e saia
5. Se dirija até a rua
6. Ande pela calçada
7. Verifique se é seguro atravessar a rua
8. Atravesse a rua
9. Continue na calçada até a venda
10. Localize o freezer
11. Localize o sorvete de creme
12. Vá ao caixa
13. Pague pelo sorvete ...

Você percebe que todos esses passos poderiam ser resumidos em tarefas de mais "alto nível"? Principalmente, se a conversa acontece entre duas pessoas que já compraram o tal sorvete na tal venda perto de casa, isso poderia causar a perda de paciência do ouvinte. Veja que podemos definir um programa "ComprarSorvete" com alguns poucos passos:

1. Pegue telefone, carteira e chave
2. Vá à venda mais próxima
3. Compre o sorvete de creme

Veja que agora temos poucas linhas definindo o que precisamos fazer. Se houver alguma dúvida em como chegar à venda mais próxima, estes são detalhes específicos. Ou seja, voltando ao nosso estudo de programação, devem estar num método específico.

*Uma dica para resolver este problema é: Se você consegue extrair um trecho do seu método para outro método, então você deveria fazer.*

*Quando estudarmos sobre estruturas condicionais, laços e tratamentos de exceções, você deveria considerar cada uma dessas coisas com a possibilidade de extrair para um método específico.*

Controle de Fluxo

Em Java, temos blocos especiais que controlam o fluxo de execução do nosso programa. Ou seja, esses blocos podem ser executados sim/não de acordo com uma condição definida (if-then, if-then-else, switch) ou ainda, podem ser executados repetidamente (for, while, do-while) de acordo com o controle definido.

Estruturas Condicionais

A estrutura mais simples é o bloco if-then. O conteúdo deste bloco será **executado apenas quando a condição definida for true**. No seguinte trecho de código, observamos uma representação de uma funcionalidade de frenagem de uma bicicleta. Assim, se o valor de isMoving for verdadeiro, então a velocidade currentSpeed será decrementada.

void applyBrakes() {

// the "if" clause: bicycle must be moving

if (isMoving){

// the "then" clause: decrease current speed

currentSpeed--;

}

}

No exemplo anterior, consideramos apenas se o valor de isMoving for verdadeiro. Mas, e quando for necessário tratar o caso contrário? Para isso, temos a palavra-chave else. Logo, temos uma estrutura chamada if-then-else. Fazendo a alteração no código anterior, temos

void applyBrakes() {

if (isMoving) {

currentSpeed--;

} else {

System.err.println("The bicycle has already stopped!");

}

}

Veja que, caso o valor de isMoving seja falso, então será exibida uma mensagem de erro no console.

Estruturas if-then-else podem ser encadeadas usando a combinação de palavras-chave else if. Veja no próximo exemplo o algoritmo para converter uma nota de 0-100 em graduações A-F. A saída esperada é Grade = C.

class IfElseDemo {

public static void main(String[] args) {

int testscore = 76;

char grade;

if (testscore >= 90) {

grade = 'A';

} else if (testscore >= 80) {

grade = 'B';

} else if (testscore >= 70) {

grade = 'C';

} else if (testscore >= 60) {

grade = 'D';

} else {

grade = 'F';

}

System.out.println("Grade = " + grade);

}

}

Existe um cenário especial, semelhante ao código anterior, quando desejamos realizar determinado comportamento de acordo com o valor contido em uma variável, sendo esse valor pertencente a um universo bem limitado. Por exemplo, desejamos converter valores 1-12 em texto com o respectivo nome do mês.

public class SwitchDemo {

public static void main(String[] args) {

int month = 8;

String monthString;

switch (month) {

case 1: monthString = "January";

break;

case 2: monthString = "February";

break;

case 3: monthString = "March";

break;

case 4: monthString = "April";

break;

case 5: monthString = "May";

break;

case 6: monthString = "June";

break;

case 7: monthString = "July";

break;

case 8: monthString = "August";

break;

case 9: monthString = "September";

break;

case 10: monthString = "October";

break;

case 11: monthString = "November";

break;

case 12: monthString = "December";

break;

default: monthString = "Invalid month";

break;

}

System.out.println(monthString);

}

}

Neste último exemplo, usamos a estrutura de controle condicional switch. A estrutura switch permite variáveis dos tipos primitivos byte, short, char e int. Também aceita Enum e String.

Note que neste exemplo simples, estamos associando um valor em monthString para cada valor de month. No entanto, imagine que desejamos saber quantos dias existem em um determinado mês. É um cenário onde temos a mesma resposta para mais de uma entrada. A estrutura switch também dá suporte a este caso, e o código seria semelhante a

class SwitchDemo2 {

public static void main(String[] args) {

int month = 2;

int year = 2000;

int numDays = 0;

switch (month) {

case 1: case 3: case 5:

case 7: case 8: case 10:

case 12:

numDays = 31;

break;

case 4: case 6:

case 9: case 11:

numDays = 30;

break;

case 2:

if (((year % 4 == 0) &&

!(year % 100 == 0))

|| (year % 400 == 0)) // checking for leap year

numDays = 29;

else

numDays = 28;

break;

default:

System.out.println("Invalid month.");

break;

}

System.out.println("Number of Days = " + numDays);// expected result is 28 (for year 2021)

}

}

Materiais complementares

* <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html>

Referências

* <https://docs.oracle.com/javase/tutorial/java/index.html>
* <https://www.devmedia.com.br/controle-de-fluxo-parte-l/5527>

# Estruturas de repetição

Java possui duas estruturas de controle de repetições: for e while (do-while é uma variação desta última). Enquanto a primeira executa uma quantidade pré-definida conhecida de repetições, a outra define repetições baseadas em uma condição booleana.

## **While/Do-while**

A estrutura while executa continuamente um bloco de código baseado na avaliação de uma expressão booleana (verdadeiro/falso). Sua sintaxe-base é

while (expression) {

// statement(s)

}

Assim como fornecemos uma variável booleana para a estrutura condicional if, aqui também podemos fornecer uma única variável ou uma expressão completa. Considere o trecho a seguir,

class WhileDemo {

public static void main(String[] args){

int count = 1;

while (count < 11) {

System.out.println("Count is: " + count);

count++;

}

}

}

A saída esperada para a execução do programa é

Count is: 1

Count is: 2

Count is: 3

Count is: 4

Count is: 5

Count is: 6

Count is: 7

Count is: 8

Count is: 9

Count is: 10

Você conseguiu "ver"? Declaramos uma variável int count com o valor inicial 1. Assim, a instrução em seguida é a estrutura de repetição condicional while. Como o valor de count é 1, e sabemos que 1 < 11 é verdade, então o bloco será executado a primeira vez, escrevendo em console Count is: 1. Em seguida, a variável count será incrementada, encerrando a primeira execução do bloco. Como estamos estudando uma estrutura de repetição, o bloco será repetido **enquanto** a expressão for verdade. Para o novo valor de count = 2, a expressão continua sendo válida, portanto, será escrita a linha Count is: 2 e assim por diante até que, após escrever a linha Count is: 10 o valor de count será incrementado, resultando 11. Na avaliação seguinte, a expressão 11 < 11 é falsa, encerrando as repetições do bloco while e, consequentemente, nosso programa.

Mas, o que acontece quando não temos um valor inicial? Esse é o caso especial da variação do-while. Enquanto no caso padrão da estrutura while a primeira execução já está sujeita a avaliação da expressão booleana (ou seja, o bloco pode ser executado 0..\* repetições). Temos que para a variação do-while o bloco será executado no mínimo 1 vez, visto que a avaliação da expressão de repetição será apenas ao final do bloco.

class DoWhileDemo {

public static void main(String[] args){

int count = 1;

do {

System.out.println("Count is: " + count);

count++;

} while (count < 11);

}

}

## **Laço de repetição FOR**

A estrutura de repetição for tem a intenção de percorrer um conjunto definido de valores. O laço for executa a repetição de um bloco de código repetidas vezes **até** satisfazer uma determinada condição. A sintaxe básica é

for (inicialização; terminação; incremento) {

instrução(ões)

}

* A expressão inicialização inicia o laço; é executado uma vez, quando o laço inicia.
* Quando a expressão de terminação é avaliada como falsa, então o laço encerra.
* A expressão de incremento é invocada depois de cada interação ao longo do laço. É perfeitamente aceitável para a expressão também conter o decremento de um valor.e.

Com base nos exemplos vistos na seção anterior (while), podemos re-escrever um programa que escreva em console todos os números inteiros entre 1-10.

class ForDemo {

public static void main(String[] args){

for(int i=1; i<11; i++){

System.out.println("Count is: " + i);

}

}

}

*Nota: É comum o uso de variáveis i, j, k como variáveis de controle do laço for.*

Os laços for são especialmente úteis para percorrer arrays. Considere o seguinte exemplo

class ForArrayDemo {

public static void main(String[] args){

int[] numbers = {1,2,3,4,5,6,7,8,9,10};

for(int i=0; i < numbers.length; i++){

System.out.println("Array value is: " + numbers[i]);

}

}

}

Veja que usamos a variável de controle i como índice para percorrer nosso array. Perceba também como definimos a condição de término do laço: i < numbers.length. Ou seja, numbers.length é uma propriedade especial de todo array que indica o seu tamanho. Assim, o valor esperado é 10, isto é, o tamanho do array criado.

## **Sintaxe especial do for para arrays**

Java também oferece uma variação do laço for, chamada **foreach**. Embora a sintaxe seja diferente, continuamos usando a palavra-chave for. Veja a seguir

class EnhancedForDemo {

public static void main(String[] args){

int[] numbers = {1,2,3,4,5,6,7,8,9,10};

for (int item : numbers) {

System.out.println("Count is: " + item);

}

}

}

Neste trecho não temos uma variável de controle. O próprio compilador Java entende que estamos percorrendo um array, e **associa cada valor contido no array à variável informada**. Ou seja, o código anterior é equivalente a

class EnhancedForDemo {

public static void main(String[] args){

int[] numbers = {1,2,3,4,5,6,7,8,9,10};

for (int i=0; i < numbers.length; i++) {

int item = numbers[i];// here is the trick!

System.out.println("Count is: " + item);

}

}

}

### Materiais complementares

* <https://docs.oracle.com/javase/tutorial/java/index.html>

### Referências

* <https://docs.oracle.com/javase/tutorial/java/index.html>
* <https://www.treinaweb.com.br/blog/estruturas-condicionais-e-estruturas-de-repeticao-em-java#:~:text=Estruturas%20de%20repeti%C3%A7%C3%A3o%2C%20tamb%C3%A9m%20conhecidas,o%20for%20e%20o%20while>.

# Arrays

Um array é um contêiner de objetos que armazena uma quantidade limitada de elementos de um único tipo. O tamanho do array é definido em sua declaração. Para declarar um array de 10 posições de números inteiros, podemos escrever

int[] arrayInteiros = new int[10];

#Uma imagem contendo relógio

Descrição gerada automaticamente

Cada elemento do array deve ser acessado através de um índice. Por exemplo, arrayInteiros[0] acessa o valor do primeiro elemento no array, ou seja, o índice inicia de 0 e vai até length - 1. Assim, para um array de 10 posições, temos os índices de 0 a 9.

Imagine que queremos gravar a idade de 10 pessoas e, após todas as leituras, queremos exibir todos os valores recebidos.

public class Main {

public static void main(String[] args) {

// declares an array of integers

int[] ages;

// allocates memory for 10 integers

ages = new int[10];

// initialize first element

ages[0] = 10;

// and so forth

ages[1] = 15;

ages[2] = 22;

ages[3] = 35;

ages[4] = 37;

ages[5] = 17;

ages[6] = 21;

ages[7] = 19;

ages[8] = 26;

ages[9] = 28;

// Alternatively, you can use the shortcut syntax to create and initialize an array:

// int[] anArray = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};

System.out.println("Element at index 0: " + ages[0]);

System.out.println("Element at index 1: " + ages[1]);

System.out.println("Element at index 2: " + ages[2]);

System.out.println("Element at index 3: " + ages[3]);

System.out.println("Element at index 4: " + ages[4]);

System.out.println("Element at index 5: " + ages[5]);

System.out.println("Element at index 6: " + ages[6]);

System.out.println("Element at index 7: " + ages[7]);

System.out.println("Element at index 8: " + ages[8]);

System.out.println("Element at index 9: " + ages[9]);

}

}

## **Strings como arrays de caracteres**

Temos que ter em mente que, assim como o tipo char representa um único carácter Unicode, String representa uma sequência de caracteres, ou seja, um **array de char**. Portanto, podemos manipular qualquer string como um char[].

Para nossa conveniência, a linguagem Java já nos fornece diversos métodos para manipulação de arrays. Vamos analisar o exemplo a seguir.

class ArrayCopyOfDemo {

public static void main(String[] args) {

char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',

'i', 'n', 'a', 't', 'e', 'd'};

char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

System.out.println(new String(copyTo));

}

}

Temos um array de origem com 13 elementos do tipo char. Desejamos obter uma cópia iniciando no elemento de índice 2 até (mas não incluindo) o índice 9. O resultado vamos atribuir a um novo array com nome copyTo. Para exibir o array resultante, vamos iniciar uma nova string com o conteúdo de copyTo e exibi-la através de System.out.println.

A saída esperada do programa deve ser caffein.

### Materiais complementares

* <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html>

### Referências

* <https://docs.oracle.com/javase/tutorial/java/index.html>

# Matrizes

Aprendemos no tópico anterior sobre arrays

*Um*array*é um contêiner de dados que armazena uma quantidade limitada de elementos de um único tipo*

Assim, nós sabemos que podemos declarar arrays de todos os tipos de dados suportados pelo Java - desde os tipos primitivos até os mais complexos, como Strings. Mas se um array é um tipo, então podemos declarar "arrays de arrays".

Vamos pensar numa planilha. Ela tem linhas e colunas. é comum que as colunas sejam letras e as linhas, números. Assim, nós referenciamos a primeira de todas as "células" como **A1**. Às células na mesma linha chamamos B1, C1, D1, etc. Às células na mesma coluna de A1 chamamos A2, A3, A4, etc. Veja que temos um padrão para referenciar células de acordo com as linhas e colunas que elas pertencem.

Se analisarmos do ponto de vista da geometria, existem espaços unidimensionais, bi-dimensionais, tri-dimensionais (3D), etc. Em plano cartesiano bi-dimensional, referenciamos as coordenadas através da posição no eixo X e no eixo Y. Da mesma forma, se for um espaço 3D, temos três coordenadas: X, Y e Z.

Com base nisso, podemos falar na programação sobre **matrizes**, ou seja, são "arrays de arrays". A sintaxe em Java é bem simples, usando os colchetes para cada dimensão que queremos representar. O trecho a seguir, cria uma matriz (ou array bi-dimensional) de 5 linhas e 5 colunas.

int[][] matrizInteiros = new int[5][5];

Veja que usamos duas vezes a notação de colchetes; uma para representar as linhas e outra para as colunas. Vamos a um exemplo usando a notação resumida de arrays para criar uma matriz 3x3 com os números inteiros de 1 a 9.

int[][] matrizInteiros = {{1,2,3},{4,5,6},{7,8,9}};

Para referenciar a primeira posição, faremos como nos demais sistemas multidimensionais discutidos anteriormente. Ou seja, a primeira posição é indicada pela primeira linha e primeira colunas, isto é, [0][0].

System.out.println(matrizInteiros[0][0]);//expected result -> 1

E para percorrer cada uma das posições da nossa matriz, precisaremos aninhar laços!

## **Laços aninhados**

Como vimos, matrizes podem ser chamadas de "arrays de arrays". Isso permite os mais variados cenários de multi-dimensões.

Quando estudamos arrays, vimos que cada posição é referenciada por um índice e a forma mais prática de percorrer todas as posições é usando um laço FOR. No caso das matrizes, vamos precisar de um laço para cada dimensão. Ou seja, para uma matriz (2D) precisaremos de 2 laços; para uma matriz 3D, 3 laços; e assim por diante.

int[][] matrizInteiros = {{1,2,3},{4,5,6},{7,8,9}};

for (int i=0; i < matrizInteiros.length; i++) {

for (int j=0; j < matrizInteiros[i].length; j++) {

System.out.println(matrizInteiros[i][j]);

}

}

### Materiais complementares

* <https://www.devmedia.com.br/matrizes-aprenda-a-trabalhar-com-vetores-bidimensionais-revista-easy-java-magazine-22/25766>

### Referências

* <https://docs.oracle.com/javase/tutorial/java/index.html>
* <http://www.w3big.com/pt/java/java-array.html>

## **Versionamento de código**

O versionamento de código é uma estratégia utilizada para gerenciar as diferentes versões de um código.

E quando falamos de versionamento de código, é comum ouvirmos falar de Git.

### Git

Git é um sistema de código aberto para o controle de versões. Com ele podemos criar todo histórico de alterações no código do nosso projeto e facilmente voltar para qualquer ponto para saber como o código estava naquela data.

### GutHub

O Github é um serviço online de hospedagem de repositórios Git(como são chamados os projetos que utilizam Git). Com ele podemos manter todos nossos commits e ramos sincronizados entre os membros do time. Além do github existem outros serviços como por exemplo o GitLab e Bitbucket.

Para utilizar o Git precisamos:

* fazer o download dele (<https://git-scm.com/downloads>)
* Criar uma conta em serviço online de hospedagem de repositórios Git.

Apos isso, devemos configurar nossa conta localmente.

$ git config --global user.name "Fulano de Tal"

$ git config --global user.email fulanodetal@exemplo.br

### Branch

As ramificações ou branchs no Git são formas de termos uma mesma versão do código sofrendo alterações e recebendo commits de diferentes fontes e inclusive por diferentes desenvolvedores.

Criando repositório novo

$ git init <meu\_projeto>

Clonando repositório

$ git clone <repo url>

Criando uma nova branch

$ git checkout -b <branch\_nova>

Adicionando nova alteração

$ git add \*

Commitando alteração

$ git commit -m “ meu commit”

Subindo commit

$ Git push origin <sua\_branch>

### Materiais complementares

* <https://www.conventionalcommits.org/en/v1.0.0/>

### Referências

* <https://git-scm.com/doc>

## **Tratamento de erro**

Quando criamos projetos Java, podem ocorrer erros durante a execução do seu projeto. Nesse momento, você já deve ter se deparado com alguns desses erros que param a execução do nosso projeto e lançam a famosa "Exception".

As Exceptions acontecem quando algo não desejado acontece. Mas, não queremos que esse imprevisto pare toda a execução do nosso projeto. Podemos contornar essas situações realizando o tratamento dos locais no código que podem vir a lançar possíveis exceções. Uma forma de fazer isso é utilizando o try e catch

Sintaxe

try{

//codigo

}catch(exceçao e){

//codigo

}

* try: Inserimos o trecho de código que queremos executar e que pode lançar uma exceção
* catch: Trecho de código que deve ser executado caso ocorra um erro e a exceção seja capturada.

Quando utilizamos o try e catch podemos adicionar o finally que é um bloco de código que será sempre executado, independentemente de sua conclusão ter ocorrido normalmente ou ter sido interrompida.

Sintaxe

try{

//codigo

}catch(exceçao e){

//codigo

}finally {

// bloco de codigo

}

### Materiais complementares

* <https://docs.oracle.com/javase/tutorial/essential/exceptions/try.html>

### Referências

* <https://docs.oracle.com/javase/tutorial/java/index.html>
* <https://www.devmedia.com.br/tratando-excecoes-em-java/25514>