Java Básico

Java

A linguagem de programação surgiu em 1991 na Sun Microsystems, em um projeto chamado Green. O objetivo era criar uma linguagem de programação 100% voltada ao paradigma de orientação a objetos que seria utilizada para criar uma rede de comunicação entre eletrodomésticos.

Porém com o surgimento das redes de internet e dos protocolos de redes que utilizamos até hoje em dia acabou dispensando o projeto Green e a linguagem de programação. A linguagem de programação java foi aberta ao público e então utilizada em diversos outros projetos.

Hoje a linguagem Java é sustentada pela Oracle que comprou os direitos em 2008.

Uma Linguagem Multiplataforma

A linguagem Java foi projetada para ser multiplataforma, isso significa que um código escrito em um sistema operacional como Linux poderá ser executado em qualquer outro sistema operacional como IOS ou Windows. Isso é possível graças a JVM - Java Virtual Machine, que realiza a execução do bytecode derivado do código java.

Todo código escrito em java antes de ser executado é transformado em bytecode para ser executado pela JVM, esse processo é chamado de compilação. Durante o processo de compilação o é identificado erros de sintaxe do código para garantir que será executado sem nenhum problema.

Os erros que acontecem durante a compilação são chamados de Exceções de compilação. Existem também os erros que acontecem durante a execução do programa, esses são chamados de Exceções de Tempo de Execução (Runtime Exceptions).

Sintaxe do Java

A sintaxe de uma linguagem são as regras e padrões que devem ser seguidos para construção de uma comunicação compreensível e coerente entre um emissor e o receptor.

Com as linguagens de programação não é diferente, cada linguagem possui suas regras para que possam ser interpretadas pelo computador. No caso então o programador é o emissor de comandos e ordem e o computador é o receptor que vai interpretar e executar os comandos do emissor. Por isso tenha em mente que o computador executará exatamente o que você mandar ele executar, se está errado ou não é da sua responsabilidade como programador.

Chaves

O Java utiliza blocos de execução para separar e organizar quais comandos devem ser executados na ordem correta, para demarcar esses blocos de execução devemos usar chaves ({ }), a todo momentos que podemos notar um chave se abrindo podemos considerar que um bloco de execução foi aberto, e sempre que notamos uma chave fechando podemos considerar que o contexto daquele bloco de execução foi encerrado.

Exemplo:

Um exemplo bem como é o método main que é responsável por executar todos o código Java escrito dentro dele.

```
public class App {

  public static void main(String[] args) {

  }

}
```

Podemos notar que logo após o nome da classe APP temos uma chave abrindo o bloco de execução da classe APP. Dentro do contexto de execução dessa classe temos o método main que também possui sua chave abrindo e por fim duas chaves fechando, uma para o método main e outra para a classe APP. Com essa leitura vemos que o método main pertence ao contexto de execução da classe APP.

Parênteses

Os parênteses no java servem para que possamos passar argumentos, parâmetros ou recursos para um método funcionar.

```
if(1 > 2){

}
```

No exemplo abaixo vemos um simples if que vai verificar se 1 é maior que 2, para poder perguntar isso vemos passar como parâmetros o número 1 o operador de comparação de maior (>) e o número 2 tudo isso dentro dos parênteses.

Os parênteses também são usados para dar ordem de prioridade para cálculos da mesma forma que na matemática convencional.

Exemplo:

```
double resultado = 2 + 3 * (3 - 2);
```

No exemplo acima notamos que vamos armazenar em uma variável o resultado do cálculo. Para o Java a prioridade está na subtração presente dentro dos parênteses, então ele primeiro irá executar o cálculo de 3 menos 2 e depois multiplicar por 3 seguindo a ordem de prioridade natural da matemática.

Ponto e Vírgula - ou Ponto sobre uma Vírgula

O ponto e vírgula para o Java funciona como ponto final para cada linha de execução, tenha muita atenção nesse momento, pois linhas de execução são diferentes de bloco de execução.

As linhas de execução são as linhas que executam comandos diretos como criar um variável ou chamar um método específico. Usamos o ponto e vírgula a todo momento para encerrar essas linhas de execução e vale lembrar que o ponto e vírgula é usado **EXCLUSIVAMENTE PARA LINHAS DE EXECUÇÃO.**

Exemplo:

```
public static void main(String[] args) {
  int numero = 2 - 1;
  System.out.println(numero);

}
```

Acima notamos que dentro do bloco de execução do método main existem 2 linhas de execução; uma para armazenar a conta 2 menos 1 em uma variável número e outra para mostrar o resultado no console utilizando o método print. Ambas as linhas terminam com ponto e vírgula para indicar a finalização de cada uma dessas linhas.

Variáveis

Durante a programação precisamos armazenar valores importantes de deverão ser “lembrados” ou “resgatados” em algum momento. Para isso existem as variáveis, com elas podemos armazenar um valor dentro de um pequeno espaço da memória ram do computador e resgatar esse valor a qualquer momento durante a execução do código.

Uma das mais famosas características do Java é sua alta tipagem para variáveis. Isso significa que no momento que criamos uma variável devemos definir se ela vai armazenar um valor do tipo texto, número, booleano, lista etc.

Exemplo:

```
public static void main(String[] args) {
  int numero;
  double numeroQuebrado;
  String texto;
  char letra;

}
```

Como podemos notar no exemplo acima 4 variáveis foram criadas dentro do método main, cada uma com seu respectivo tipo, sendo a primeira para número

inteiros representando pelo int a segunda para números que possuem vírgula representado pelo double, a terceira para textos representando pelo String e por último o char que serve para armazenar apenas 1 letra.

As tipagens são dividas em 2 grupos: os **naturais** e os **wrappers**.

Tipos Naturais

Os tipos naturais são os mais básicos na linguagem java, eles praticamente servem apenas para armazenar um valor. Uma característica importante dos tipos naturais é que variáveis tipadas com eles não podem receber NULL que é a representação de um espaço vazio.

Tipo	Descrição
boolean	Apenas valores true ou false
byte	Números inteiros de até 1 bytes
short	Números inteiros de até 2 bytes
int	Números inteiros de até 4 bytes
long	Números inteiros de até 8 bytes
double	Números com vírgula de até 8 bytes
float	Números com vírgula de até 4 bytes
char	1 única letra ou caracter

Tipos Wrappers

Os wrappers são tipagens que possuem métodos para auxiliar no desenvolvimento além de armazenar valores NULL, coisa que as tipagens naturais não permitem.

Tipo	Descrição
Boolean	Apenas valores **true** ou **false**

Byte	Números inteiros de até 1 bytes
Short	Números inteiros de até 2 bytes
Integer	Números inteiros de até 4 bytes
Long	Números inteiros de até 8 bytes
Double	Números com vírgula de até 8 bytes
Float	Números com vírgula de até 4 bytes
String	Textos inteiros
Character	Apenas 1 letra

Um detalhe importante é que os wrappers são escritos com a primeira letra maiúscula, isso acontece porque todos eles são classes do Java e são através dessas classes que podemos usar os métodos como comparar dois textos, alterar valor em uma posição específica etc.

Operador de Atribuição

No exemplo anterior nós apenas criamos as variáveis, porém para preencher esse espaço de memória criado com algum valor precisamos usar um operador de atribuição, esse operador é o final de igual da matemática (=).

Exemplo:

```
int numero = 10;
String texto;
texto = "Adeus e obrigado pelos peixes";
```

Nesse exemplo podemos notar que temos 2 variáveis criadas porém são preenchidas em tempos diferentes.

A variável **"número"** é criada e ao mesmo tempo já usamos o operador de atribuição para preenchê-la com o valor 10.

Já a segunda variável é criada em um primeiro momento e apenas na próxima linha de execução preenchemos ela com um valor.

É importante mencionar que os valores dentro de cada variável pode ser facilmente alterado a qualquer momento durante a programação caso o

programador queira, basta utilizar o nome da variável seguido do operador de atribuição e por fim o novo valor.

Estruturas de Tomada de Decisão

Uma situação bem como durante a programação é a necessidade de comparar dois valores para saber se são iguais, isso pode acontecer para comparar senhas ou então mudar uma resposta para o usuário dependendo do que ele digita para o sistema. Essas situações são feitas através de uma série de condições pré determinadas dentro do programa criado, podemos até dizer que o coração da inteligência artificial é um conjunto complexo de condições previstas pelo programador.

As estruturas de tomada de decisão são justamente para preparar essas condições. Na programação Java usamos os **if**, **else** e **else if** para tomar decisões específicas de acordo com condições pré programadas.

Exemplo:

```
int sexta = 13;

if(sexta != 13){
  System.out.println("Dia de sorte");
}else{
  System.out.println("Hoje é uma sexta-feira 13. Bruxa está solta");
}
```

No exemplo acima temos uma variável que armazena um número inteiro a condições que preparei dentro do **if** vai analisar o valor armazenado dentro da variável chamada sexta, caso o valor lá dentro seja **diferente de 13** então o programa exibirá **“Dia de sorte”**.

Nesse caso temos já existe uma possibilidade oposta a essa condição que é o número ser exatamente 13, nesse caso o **else** entra em jogo exibe o texto **“Hoje é uma sexta-feira 13. Bruxa está solta”**

Operadores de Comparação

Para realizar comparações de valores, usamos operadores capazes de dizer se uma comparação é verdadeira ou falsa, estes ponderadores literalmente vão literalmente comparar dois valores de retornar como resposta um true ou false que podemos armazenar em uma variável ou usar diretamente dentro de um if como vimos no exemplo acima.

Operador	Descrição
==	se são exatamente iguais
!=	se são diferentes
<	se um é menor que outro
>	se um é maior que outro
<=	se um é menor ou igual a outro
>=	se um maior ou igual a outro

Portas Lógicas

Além dos operadores de comparação, também podemos utilizar dentro de um if uma porta lógica para realizar mais de uma comparação ao mesmo tempo.

Exemplo:

```
int idade = 18;
boolean ingresso = true;

if(ingresso & idade >= 18){
  System.out.println("Pode Entrar");
}
```

No exemplo acima foram criadas 2 variáveis que representam 2 valores diferentes que devem ser verificados dentro de um **IF**. O primeiro valor é a idade e o segundo é um boolean dizendo se a pessoa possui um ingresso, dentro do IF vamos verificar se o valor da variável ingresso é true e se a idade é maior ou igual a

18 anos. Caso uma delas seja falsa o **IF** não deve ser executado, é por isso que usamos o sinal de **E** comercial (&) que corresponde a porta lógica **"and"**.

Com a porta lógica **AND** vamos garantir que o **IF** seja executado apenas quando as duas condições forem verdadeiras.

Existem outras portas lógicas além da **AND**, observe na tabela abaixo.

Operador	Porta	Descrição
&	and (e)	Apenas verdadeiro quando ambas as condições são verdadeiras
|	or (ou)	Apenas verdadeiro quando pelo menos uma condição for verdadeira
!	not	Inverte o resultado da saída. Exemplo: se o resultado for verdadeiro ele troca para falso. Se o resultado for falso ele troca para verdadeiro
^	xor (exclusivo ou)	Apenas verdadeiro quando um condição for false e a outra verdadeira

If e else if

Já vimos que o if é capaz de receber um valor true se ser executado, porém em alguns momentos precisamos prever diversos cenários que podem acontecer no nosso código, para auxiliar nesses momentos nós utilizamos o if junto com o else if.

Exemplo:

```
int idade = 18;

if (idade >= 18){
      System.out.println("É maior de idade");
  }else if (idade == 17){
      System.out.println("É menor de idade");
  }else if (idade < 16){
      System.out.println("É praticamente um bebe");
```

```
}
```

No exemplo acima temos três diferentes reações que o código deve fazer: uma para caso a idade for maior de 18, uma para caso a idade seja igual a 17 e a última caso a idade seja menor de 16 anos.

Sendo assim usamos o else if para criar uma estrutura inteligente que vai verificar a idade armazenada na variável idade. O comportamento do java nesse caso é de procurar o primeiro if ou else if que seja verdadeiro e executar o código, assim que ele encontra a primeira condição verdadeira o java vai executar as linhas dentro do bloco de execução e ignorar os demais else if.

Estruturas de Repetição

Em momentos na programação nós precisamos repetir diversas vezes uma mesma função ou algoritmo, nesses casos precisamos de estruturas de repetição que vão literalmente repetir em um número limitado de vezes aquele algoritmo.

Isso é extremamente vantajoso evitar a repetição desnecessária de código e para criar programação que só param sua execução quando atingem um objetivo específico.

While

Uma das estruturas de repetição é o while que poderia ser traduzido para a palavra “enquanto”. O while funciona a partir de uma condição que for verdadeira, enquanto essa condição continuar verdadeira o código dentro do contexto do while será executado,

Isso é muito importante saber, pois se essa condição nunca mudar para falso o while nunca irá parar criando assim um looping infinito.

Exemplo:

```
int contador = 0;

while (contador <= 10){
   System.out.println(contador);
   contador++;
}
```

No exemplo acima, criamos um while que espera que a condição do valor dentro da variável contador seja menor ou igual a 10, enquanto a condição for verdadeira o código dentro do bloco do while será executado, sendo assim vamos exibir na tela o valor contido dentro da variável contador e vai somar mais 1 ao valor dentro da variável.

Por fim, quando o valor dentro da variável for 11 ele a condição de menor ou igual a 10 vai ser falsa e o while vai para sua execução.

IMPORTANTE: lembre-se que o while vai funcionar apenas enquanto a condição for verdadeira (true), e só para enquanto a condição for falsa (false).

For

Uma outra estrutura de repetição que podemos usar é o **FOR** que podemos traduzir para **"para cada"**.

Embora o for tenha a mesma função do **while** de gerar repetição no código e facilitar nossa vida como programador ele tem uma maneira diferente de trabalhar.

Exemplo:

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

No exemplo acima tem um for que realizará a mesma tarefa do exemplo com while visto anteriormente, porém com o for podemos construir as condições para seu funcionamento dentro da sua própria assinatura, isso de certa forma cortamos o caminho e no final das contas temos o mesmo resultado que é contar até 10 e exibir na tela.

For each

O interessante é que com o for também podemos percorrer listas e vetores para facilitar o trabalho na hora de trabalhar com conjunto massivo de dados.

Com o for each podemos acessar separadamente cada item dentro de um vetor e trabalhar exclusivamente com esse item e partir para o próximo com facilidade.

Exemplo:

```
String[] nomes = {"Vinícius", "José", "Saulo", "Miriana",
"Diandra"};

for(String nome : nomes){
   System.out.println( nome.toUpperCase() );
}
```

No exemplo acima usamos um for each para percorrer um vetor de nomes, nesse caso o for vai pegar um nome de cada vez dentro da variável **nomes** e colocar dentro da variável **nome** assim podem trabalhar com cada um separadamente. Nesse exemplo nós estamos imprimindo na tela do usuário cada um dos nomes dentro do vetor com todas as letras maiúsculas.

Listas

Armazenar informações durante a execução do código é sempre importante e para facilitar e centralizar informações contamos com vetores, porém ele são imutáveis e não são dinâmicos, quando precisamos adicionar ou remover valores precisamos trabalhar com listas.

Exemplo:

```
List<String> nomes = new ArrayList<>();
```

As listas podem ser alteradas a qualquer momento aumentando ou diminuindo seu tamanho conforme a quantidade de dados dentro delas.

Como as listas são um tipo de objeto bem específico do java, ao usá-las ganhamos diversos métodos que nos ajudam a manipular os dados.

ADD

Esse método adiciona um novo item no final da lista.

Exemplo:

```
List<String> nomes = new ArrayList<>();
nomes.add("Vinicius");
```

Ao usar o método add o nome "Vinícius" será adicionado ao final da lista.

GET

Com método get resgatamos da lista o valor a partir da posição que queremos.

Exemplo:

```
List<String> nomes = new ArrayList<>();
nomes.get(0);
```

O método get vai resgatar o valor esse valor pode ser colocado em uma variável para ser usada posteriormente. Vale lembrar que toda lista começa na posição 0, ou seja em uma lista com 3 itens as posições são 0, 1 e 2.

MAP

Em outros casos precisamos armazenar valores de maneira mais organizada como se fosse um banco de dados onde tenho chave e valor. Para isso podemos contar com a estrutura de memória map.

Com ela podemos armazenar de maneira mais organizada e de fácil acesso os dados a partir de chave e valor.

Exemplo:

Chave	Valor

Vinicius	27 anos
Camila	28 anos

No exemplo acima pode-se notar que uma estrutura de chave e valor é muito parecido com uma planilha onde de uma lado temos a chave que é o nome de cada pessoa e do outro temos o valor que é a idade de cada um.

PUT

Com o map temos a possibilidade de adicionar uma nova chave com seu respectivo valor em um map.

Exemplo:

```
Map<String, Integer> nomeEIdade = new HashMap<>();
nomeEIdade.put("vinicius", 27);
```

É bom mencionar que no exemplo acima estamos adicionando a chave “vinicius” com o valor vinculado que é o número 27.

Uma característica do map é que a chave é única ela não podem repetir, caso seja usado um put para adicionar uma chave que já existe o valor da chave será alterado.

GET

Com o get podemos resgatar um valor a partir de uma chave fornecida.

Exemplo:

```
nomeEIdade.get("vinicius");
```