

PROGRAMAÇÃO I

Aula 2: Características da linguagem Java

Apresentação

Trabalharemos, em Java, com os principais conceitos de linguagens de programação, como variáveis, constantes e tipos de dados.

Em seguida, abordaremos operadores e expressões, estruturas condicionais e de repetição.

Ao final, desenvolveremos programas simples em Java.

Bons estudos!

Objetivos

- Analisar os tipos de dados, constantes e como declarar variáveis em Java;
- Identificar operadores e expressões da linguagem Java;
- Apontar os comandos das estruturas condicionais e de repetição;
- Desenvolver programas simples em Java.

Características da linguagem Java

A linguagem Java tem boa parte de suas características herdadas da linguagem C. Muitos dos seus operadores, formação de identificadores, comandos de controle de fluxo e várias outras características são compartilhados entre estas duas linguagens.

Todas as instruções da linguagem Java devem terminar por um símbolo de ponto e vírgula “;”. Você não usará o ponto e vírgula quando a instrução for uma codificação que irá continuar com um bloco de comandos.

Vejamos um exemplo:

```
System.out.println("Mensagem do sistema");
```

Os blocos de comandos em Java são delimitados por { (abrir) e } (fechar) chaves, em que a instrução anterior define que todos os comandos do bloco farão parte desta. Isso irá ocorrer em classes, métodos e instruções de controle de fluxo.

Exemplo:

```
if(nota>10.0) {  
    System.out.println("Nota inválida");  
}
```

Como usar a endentação?

Quando desenvolvemos um programa em qualquer linguagem, é comum que utilizemos um conjunto de espaços na frente das instruções de forma a facilitar a visualização de blocos. Sempre que iniciamos um bloco, devemos começar na próxima linha com um deslocamento de pelo menos quatro espaços em branco ou uma tabulação (normalmente quatro espaços). Isso permite que identifiquemos rapidamente que certo conjunto de instruções faz parte de um conjunto que será executado em bloco.

Exemplo:

```
import java.util.Scanner;
public class Exemplo {
    // indentação do bloco da classe Exemplo
    public static void main(String[] args) {
        // indentação do bloco do método main
        // TODO Auto-generated method stub
        Scanner sc = new Scanner(System.in);
        double media, nota1, nota2;
        System.out.println("Digite a nota 1:");
        nota1 = Double.parseDouble(sc.nextLine());
        System.out.println("Digite a nota 2:");
        nota2 = Double.parseDouble(sc.nextLine());
        media = (nota1 + nota2) / 2.0;
        System.out.println("A sua média é:" + media);
        sc.close();
    } // encerramento da indentação do bloco do método main
} // encerramento da indentação da classe Exemplo
```

Em todas as linguagens de programação, devemos identificar variáveis, programas, funções, métodos, parâmetros etc. O ato de nomear algo em uma linguagem de programação é uma forma de identificação da linguagem. Em Java são permitidos identificadores que comecem com letras (maiúsculas ou minúsculas), ou um símbolo de “\$” (dólar) ou “_” (*underscore* / *underline*). Números podem ser usados, mas não para iniciar um identificador.



Java é uma linguagem de programação sensível à caixa (alta ou baixa ou *case sensitive*). Desta forma, a linguagem faz distinção entre letras maiúsculas e minúsculas. Mas isso não quer dizer que podemos utilizar qualquer nome como um identificador, pois existem algumas palavras reservadas que não podem ser utilizadas para tal.

Exemplos de identificadores válidos em Java:

```
identificador
nomeCompleto
NomeCompleto
nota1
_sys_path
```

Observe que os exemplos 2 e 3 possuem a mesma grafia, mas, como existe mudança entre caixa alta e baixa, para a linguagem Java são dois diferentes identificadores.

Palavras reservadas da linguagem Java (não podem ser usadas como identificadores); dentre elas podemos destacar:

abstract	do	implements	private	throw
boolean	double	import	protected	throws
break	else	instanceof	public	transient
byte	extends	int	return	true
case	false	interface	short	try
catch	final	long	static	void
char	finally	native	super	volatile
class	float	new	switch	while
continue	for	null	synchronized	
default	if	package	this	


Atenção

Todas as palavras reservadas começam por letras minúsculas e são palavras do idioma inglês.

Comentários

O uso de comentários em Java é semelhante ao usado na linguagem C, mas apenas dois destes tipos são iguais nas duas linguagens, sendo o terceiro tipo somente disponibilizado na linguagem Java.

Vejamos:

 Clique nos botões para ver as informações.

1) Comentário de linha



```
// comentário de linha
```

Quando usamos duas barras em uma linha de código, todo o seu conteúdo, após as duas barras, é desconsiderado pelo compilador, o que quer dizer que podemos escrever qualquer conteúdo que o mesmo não será compilado. É muito usado para deixar informações e avisos do programador no código.

2) Comentário de bloco



```
/*
```

```
comentário de linha
```

```
*/
```

Ao usarmos o comentário de bloco, podemos comentar não apenas um trecho de uma linha, mas todo um conjunto de linhas. É utilizado quando temos longos trechos de textos com avisos e informações, ou para a depuração do código. Para a depuração do código, podemos comentar um conjunto de linhas para realizar um conjunto de testes. Neste caso, podemos comentar um conjunto de instruções ou porque estas instruções estão com problemas e queremos verificar as demais. Ou, ao contrário, onde temos um conjunto de instruções já testadas e corretas e queremos apenas testar as demais. Seja como for, o uso do comentário de bloco é muito usual e comum entre os programadores.

```
/**
```

```
comentário de documentação
```

```
*/
```

O comentário de documentação se difere do comentário de bloco por possuir um asterisco a mais no início, mas ambos encerram da mesma forma. Existe uma ferramenta na linguagem Java responsável por extrair de um projeto (com várias classes) todos os comentários de documentação e montar um documento com todo este conteúdo.

Neste caso, usamos este tipo de comentário apenas para descrever avisos e informações das classes, de forma a realizar a documentação do sistema ainda durante sua fase de criação. Isto permite que o desenvolvedor descreva toda a documentação no próprio projeto, facilitando a descrição e a manutenção do sistema. Assim, ao terminar um projeto ou realizar algum tipo de modificação, basta gerar novamente a documentação do sistema que tudo estará atualizado.

Tipos de dados

A linguagem Java possui nove tipos de dados básicos, sendo oito deles primitivos e um tipo especial.

Primitivos (armazenam apenas valores)

Tipo lógico (*boolean*)

O tipo lógico só permite dois estados, verdadeiro (*true*) ou falso (*false*); em Java ainda é permitido o uso de *on* e *off*, ou *yes* e *no*.

Exemplo:

```
boolean status = true;
```

Tipo caractere (*char*)

O tipo *char* permite que seja armazenado na memória apenas um caractere e se difere do texto (*String*) por ser definido entre *'*. Quando usamos aspas simples ou plica determinamos apenas um caractere.

Exemplo:

```
char letra = 'A';
```

Armazenamento de caracteres de controle

Também é possível armazenar caracteres de controle:

Caractere Especial	Representação
<code>'\n'</code>	novalinha
<code>'\r'</code>	enter
<code>'\u????'</code>	especifica um caractere Unicode o qual é representado na forma Hexadecimal.
<code>'\t'</code>	tabulação
<code>'\\'</code>	representa um caractere <code>\</code> (barra invertida)
<code>'\"'</code>	representa um caractere <code>"</code> (aspas)

Atenção

A barra invertida na frente indica que é um caractere especial.

Tipos inteiros (*byte*, *short*, *int* e *long*)

São quatro diferentes tipos de inteiros, que se diferenciam pela quantidade de *bits* que cada um ocupa em memória para armazenar um valor. Isto faz com que, quanto menor a quantidade de *bits*, maior seja a limitação do valor a ser armazenado. Entretanto, em ocasiões onde a memória é pouca, devemos trabalhar muito bem com estas diferenças para reduzir o espaço de memória necessário. O uso mais comum é do *int*, mas, para números muito grandes ou muito pequenos, devemos usar o *long*. Já para economizar memória podemos usar *byte* ou *short*, de acordo com o valor que será armazenado.

Tipo de dado	Quantidade de bits	Quantidade de Bytes	Escopo (valores que podem ser armazenados)
byte	8	1	$-2^7 \dots 2^7 - 1$
short	16	2	$-2^{15} \dots 2^{15} - 1$
int	32	4	$-2^{31} \dots 2^{31} - 1$
long	64	8	$-2^{63} \dots 2^{63} - 1$

Tipos reais (*float* e *double*)

São dois diferentes tipos de valores reais, sendo um de precisão simples (*float*), que ocupa menos espaço de memória, e o de dupla precisão, que ocupa mais memória. Quanto maior o número de bits para armazenar um valor real, maior será a precisão deste número dentro do sistema. O uso do *float* é comum quando necessitamos economizar espaço de memória. Em Java, todo tipo de dado numérico é convertido para *double* automaticamente por coerção (força a conversão de tipo). Por isso, é mais indicado, quando não houver falta de espaço de memória, a utilização de *double* para armazenamento de valores reais.

Tipo de dado	Quantidade de bits	Quantidade de Bytes
float	32	4
double	64	8

Tipo especial

Tipo texto (*String*)

O tipo texto (*String*) não é um tipo primitivo, mas um tipo especial. Na verdade, o tipo *String* é uma classe e por isso começa com letra maiúscula, ao contrário dos tipos primitivos, que sempre começam por minúsculas. Este tipo de dado armazena um conjunto de caracteres, formando palavras ou frases de tamanhos variados. Como classe, veremos mais tarde que elementos do tipo *String* possuem métodos que podem realizar ações específicas sobre o seu conteúdo.

Exemplo:

```
String nome = “João da Silva”;
```

Constantes e variáveis

Variáveis e constantes em Java devem obrigatoriamente possuir um tipo. Isso ocorre porque Java é uma linguagem de programação fortemente *tipada*.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Linguagens de programação fortemente tipadas

Obrigam que todas as variáveis e constantes sejam definidas por um tipo de dado.

Linguagens de programação fracamente tipadas

Permitem que variáveis sejam usadas a qualquer momento, sem a necessidade de terem um tipo predefinido. Isso quer dizer que o tipo de dado pode variar em diferentes partes do programa.

Variáveis são declaradas por meio de um tipo e um identificador, sem que sejam necessárias outras informações. A boa prática em programação Java determina que todas as variáveis comecem por letras minúsculas e, somente se tiver mais de uma

palavra, o inicial da segunda palavra em diante deverá começar por letras maiúsculas.

Exemplos:

```
int c;
double nota1 = 0; // indica que a variável será inicializada com 0 (zero)
String nomeCompleto;
```

A definição de constantes precisam do modificador final, que indica que, uma vez que ocorreu uma atribuição a variável, seu conteúdo não poderá ser mudado. Em Java, constantes podem ser criadas em nomes em minúsculas ou maiúsculas, mas a boa prática de programação determina que sua identificação deve ser toda em maiúsculas.

Exemplos:

```
final int IDADEMINIMA = 15;

final double VALORDOLAR = 3.96;

final NOMEEMPRESA = “Super Empreendimentos”;
```

Operadores e expressões

Operadores aritméticos

Operador	Descrição
=	Atribuição
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Restoda divisão inteira (ambos os operandos devem ser inteiros

Atenção

Java sempre realizará a operação inteira quando os operandos forem inteiros, e a operação real ocorrerá caso um ou mais operando seja real.

Exemplos:

```
int v = 7 / 2; // o valor atribuído será 3 e não 3.5, porque ambos os operandos são
// inteiros
double v = 7.0 / 2; // o valor atribuído será 3.5, porque o primeiro operando é real.
```

A mesma lógica serve para variáveis:

```
int a, b=7, c=2;
a = b / c; // será armazenado 3 em a.
double a, b=7.0, c=2.0;
a = b / c; // será armazenado 3.5 em a.
```

Podemos alterar o tipo de um operando em uma expressão utilizando o `cast`, que nada mais é do que informar que o valor armazenado na variável terá o seu valor em função do tipo alterado.

Exemplo:

```
int b=7, c=2;
double a=0;
a = (double) b / c; // o valor de b será convertido para double antes da operação
// e isso fará com que o primeiro operando seja real e desta
// forma a operação será real, armazenado 3.5 em a.
```

Operadores aritméticos

```
+=    -=    *=    /=    %=
```

Exemplo:

```
int alturaParede = 2.85; // declaração da variável alturaParede
alturaParede += 0.15; // a variável alturaParede terá o valor
                        // acrescido (somado) em 0.15, sendo
                        // equivalente a:
                        // alturaParede = alturaParede + 0.15;
```

Desta forma, não precisamos colocar o nome da variável duas vezes.

Operadores de Incremento e decremento

Em Java temos os operadores de incremento ++ e de decremento --, que sempre adicionam uma unidade (++) ou subtraem uma unidade (--). Eles podem ser ainda divididos em pré-incremento e pós-incremento, e pré-decremento e pós-decremento.

O pré-incremento determina que primeiro seja realizada a operação de incremento e depois é realizada a operação de atribuição.

Exemplo:

```
int a = 20, b=0;
b = ++a; // primeiro a variável a será incrementada de uma unidade, valendo 21,
          // depois b receberá o valor de a e assim, também valerá 21.
```

O pós-incremento determina que antes seja realizada a atribuição para só então ser realizada a operação de incremento.

Exemplo:

```
int a = 20, b=0;
b = a++; // primeiro b receberá o valor de a, que é 20 (antes do incremento),
          // depois a será incrementado e assim, o valor de a será 21 e o de b será 20.
```

O pré-decremento determina que primeiro seja realizada a operação de decremento e depois é realizada a operação de atribuição.

Exemplo:

```
int a = 20, b=0;
b = --a; // primeiro a variável a será decrementada de uma unidade, valendo 19,
          // depois b receberá o valor de a e assim, também valerá 19.
```

O pós-decremento determina que antes seja realizada a atribuição para só então ser realizada a operação de decremento.

Exemplo:

```
int a = 20, b=0;
b = a--; // primeiro b receberá o valor de a, que é 20 (antes do incremento),
// depois a será decrementado e, assim, o valor de a será 19 e o de b será 20.
```

Operadores de Relacionais (usados para definir condições)

Operador	Descrição
==	Igualdade/ Comparação
!=	Negação
>	Maiorque
<	Menorque
>=	Maiorou igual a
<=	Menorou igual a

Exemplos:

```
1) if(a > b) { ... }
2) while (a <=100) { ... }
3) for (int c =0; c<50; c++) { ... }
```

Operadores de em Expressões Lógicas

Operador	Descrição
!	NÃO lógico
&&	E lógico
	OU lógico

São os determinantes das tabelas-verdade.

Ordem de precedência: !, &&, ||

Exemplos:

```
if(a > b && c < d) { ... }
while (a <=100 || b == 10) { ... }
if( !a == 15 && b >= 10) { ... }
if( !a == 15 || c > d && b >= 10) { ... }
```

Pela ordem de precedência: if((!a == 15) || (c > d && b >= 10))

Primeiro será executada a negação (!); depois o e lógico (&&) e por último o ou lógico (||).

Operadores de bits

Operador	Descrição
&	E entre bits
^	OU EXCLUSIVO entre bits
	OU entre bits

Ordem de precedência: &, ^, |

Exemplo

Comandos de controle de fluxo

Estruturas Condicionais

Primeira estrutura

```
Se (if):
    if (condição) {
        // instruções;
    }
// A cláusula else é opcional.
```

Segunda estrutura

```
if ... else,
    if (condição) {
        // instruções;
    }
    else {
        // instruções;
    }
```

Terceira estrutura

```
if ... else if ... else,
    if (condição1) {
        instruções;
    }
    else if (condição2) {
        instruções;
    }
    else if (condição3) {
        instruções;
    }
    else {
        instruções;
```

}

Atenção

- A cláusula `if` deve ocorrer apenas uma vez;
- As cláusulas `else if` podem ocorrer: nenhuma, uma ou várias vezes;
- A cláusula `else` só pode ocorrer uma única vez.

Quarta estrutura

`switch ... case`

Estruturas de decisão caracterizadas pela possibilidade de uma variável possuir vários valores diferentes em uma determinada situação.

Uma única estrutura *switch* pode analisar vários diferentes valores para a variável de controle. A variável de controle em Java pode ser do tipo: inteiro, caractere, ou *String*.

A cláusula `case` pode ocorrer de uma a várias vezes, e a cláusula *default* é opcional.

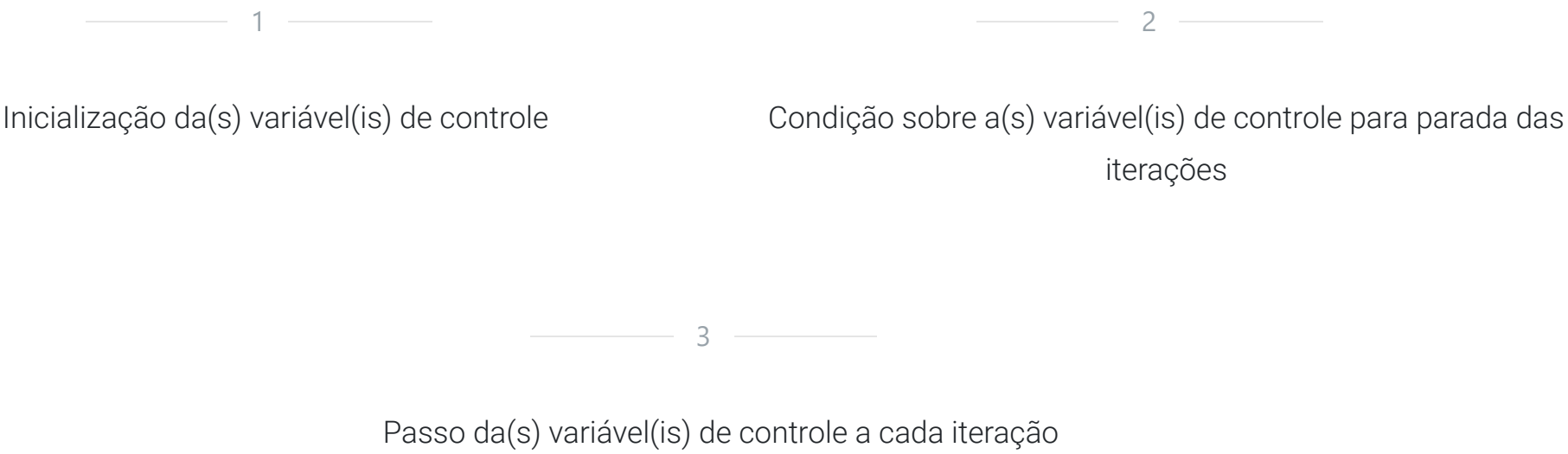
Exemplo

Estruturas de Repetição

for

Estrutura de repetição controlada por uma ou mais variáveis contadoras e caracterizada pela existência de três parâmetros, sendo todos eles opcionais:

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online



```
for((1)inicialização; (2)condição de controle; (3)passo) {  
    // instruções  
}
```

Exemplo

1) Repetição controlada por uma variável:

```
for (int c=1; c<=limite; i++) {  
    instruções;  
}
```

2) Repetição controlada por duas variáveis:

```
for (a=1, b=2; a*b<limite; a++, b+=2) {  
    instruções;  
}
```

3) Repetição sem fim

```
for ( ; ; ) {  
    instruções;  
}
```

while

Esta estrutura realiza a repetição de um conjunto de instruções enquanto a condição determinada for verdadeira; caso a condição seja falsa no primeiro teste, nenhuma instrução será executada.

```
// realiza o teste da condição no início da estrutura  
while (condição) {  
    instruções;  
}
```

do ... while

Esta estrutura de repetição é semelhante à anterior, mas com o diferencial de que as condições devem ser verificadas apenas no final da estrutura, obrigando que as instruções sejam sempre executadas pelo menos uma vez.

```
// Teste de condição no final  
do  
{  
    instruções;  
} while (condição);
```

Entrada e Saída de dados

Entrada de dados

Em Java temos muitas formas de entrada de dados, inclusive de forma gráfica. Inicialmente trabalharemos com a classe Scanner, responsável pela entrada de dados em formato texto, com perguntas diretas ao usuário e a inclusão da resposta em variáveis do programa.

Para realizarmos esta tarefa, é necessário que seja criado um objeto da classe Scanner.

Para isso, devemos importar a classe Scanner antes do início da programação da classe:

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

```
import java.util.Scanner;
```

Depois é necessário criar o objeto para realizar as entradas de dados:

```
public class EntradaDados {  
    public static void main(String[] args) {  
        Scanner entrada = new Scanner(System.in);  
    }  
}
```

Existem vários métodos associados a classe Scanner para a entrada de dados, mas para evitarmos problemas futuros podemos usar sempre a entrada de dados de texto (nextLine()) e converter o texto para o tipo desejado.

Exemplo

```
1) Para entrada de texto (String):
String nome;
Nome = entrada.nextLine(); // não precisa de conversão, apenas da entrada.
```

```
2) Para entrada de valor real:
double nota1;
nota1=Double.parseDouble(entrada.nextLine());
// a entrada de dados em texto precisa de conversão para double.
```

```
3) Para entrada de valor inteiro:
int idade;
idade = Integer.parseInt(entrada.nextLine());
// a entrada de dados em texto precisa de conversão para int.
```

É aconselhável evitar o uso de métodos como:

_____ 1 _____

entrada.nextDouble();

_____ 2 _____

entrada.nextFloat();

_____ 3 _____

entrada.nextInt();

Estes métodos, quando usados em conjunto, podem fazer com que a aplicação pule alguma entrada de dados, sendo necessário que seja realizada uma “limpeza de buffer”. Este tipo de problema pode ser contornado ao usar sempre o método “nextLine()” e a conversão de tipos.

Saída de dados

A saída de dados em modo texto pode ser realizada pela classe System, e o método out.print (não pula linha), out.println (pula linha) ou outros métodos:

1) Apenas uma mensagem:

```
System.out.println("Entre com a Nota A1.....: ");
```

2) Mensagem e conteúdo de variáveis:

```
System.out.println(" Nome: " + nome + " Idade: " + idade+ " Nota 1: " + nota1);
```

A seguir temos o exemplo completo de um programa que recebe duas notas e apresenta a média.

Exemplo

```
public class Exemplo {
```

```
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Scanner sc = new Scanner(System.in);
        double media, nota1, nota2;
        System.out.println("Digite a nota 1:");
        nota1 = Double.parseDouble(sc.nextLine());
        System.out.println("Digite a nota 2:");
        nota2 = Double.parseDouble(sc.nextLine());
        media = (nota1 + nota2) / 2.0;
        System.out.println("A sua média é:" + media);
        sc.close();
    }
}
```

Conversão de tipos

A conversão de tipos em Java pode ser feita por *cast* ou com o uso de conversão por classes. Ambos já foram vistos em exemplos anteriores:

1) Por *cast*:

Usado para converter valores de um tipo para outro; com *cast* basta indicar o tipo que você quer que a variável, ou valor, seja convertida, colocando o tipo desejado entre parênteses:

```
int a = 10;
double b = 0;
b = (double) a;
a = (int) b;
```

2) Por uso de classes para conversão de textos em valores

```
double nota1 = Double.parseDouble("7.8");
int idade = Integer.parseInt("34");
float valor = Float.parseFloat("2.15");
long valor2 = Long.parseLong("3456789");
Pode-se usar ainda:
Byte.parseByte() / Short.parseShort()
```

A boa prática em programação Java (BP)

A boa prática em programação Java leva em conta um conjunto de regras que facilitam o desenvolvimento de aplicações e melhoram bastante o trabalho em conjunto realizado por equipes. Ao seguir estas regras, projetos podem ser desenvolvidos em paralelo por diferentes programadores, sem que seja necessário que cada componente precise esperar que outros terminem suas tarefas. Estas regras foram utilizadas na construção da linguagem permitindo que não seja necessário decorar as sintaxes de instruções Java. Não são obrigatórias, mas permitem a codificação melhor de nossas aplicações.

Regras:

1) Variáveis auxiliares, atributos, métodos e objetos devem ser identificados iniciando por letras minúsculas. Quando houver mais de uma palavra, deve-se começar cada nova palavra com uma letra maiúscula.

Exemplos:

```
int idade;
int maiorIdade;
String nome;
String nomeCompleto;
```

2) Constantes devem ser identificadas por letras maiúsculas em todo o seu nome; mesmo quando temos mais de uma palavra, todo o identificador deve ficar em maiúsculas.

Exemplos:

```
final int idade;
final int maiorIdade;
final String nome;
final String nomeCompleto;
```

3) Classes e interfaces (tipo especial de classe) devem iniciar por letras maiúsculas. Quando houver mais de uma palavra, deve-se começar cada nova palavra com uma letra maiúscula.

Exemplos:

```
public class Carro { ... }
public class Carro Hibrido { ... }
public interface Basico { ... }
public interface MetodosBasicos { ... }
```

Atividade

1 - Podemos dizer que variáveis são um espaço de memória que reservamos e nomeamos. Assinale a afirmativa que apresenta um identificador válido em Java:

- a) final
- b) a&b
- c) short
- d) \$test
- e) 1test

2 - Durante o desenvolvimento de um programa em Java, o programador precisa armazenar a idade de uma pessoa (em anos). Qual o tipo de dados ideal (de forma a ocupar menos bytes de memória) a ser usado?

- a) long
- b) byte
- c) int
- d) float
- e) short

3 - Assinale a alternativa que apresenta APENAS os exemplos de operadores lógicos em Java.

- a) <, >, >=
- b) &&, ||, !
- c) =, >, ||
- d) &&, >=, , ||
- e) +, -, *

4 - Indique a saída fornecida pelo trecho de programa Java abaixo:

```
int a=2, b=5, c=4, d=3, x;  
if( d<=2 && a*2 > d+b)  
    { x = a + c * b + c; }  
else  
    { x = a + b - c * 2; }
```

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

- a) -1
- b) 6
- c) 54
- d) 26
- e) 4

Notas

Referências

Próxima aula

- Classes e objetos.

Explore mais

Pesquise na internet sites, vídeos e artigos relacionados ao conteúdo visto.

Em caso de dúvidas, converse com seu professor online por meio dos recursos disponíveis no ambiente de aprendizagem.