EAGS SIN 2022

JAVA – EAGS SIN 2022

Conteúdo do Edital:

- Estrutura.
- Variáveis.
- Classes.
- Atributos.
- Métodos.
- Herança.
- Polimorfismo.
- Encapsulamento.
- Estrutura de Controle.
- Recursividade.

ESTRUTURA E VARIÁVEIS

Introdução a Programação Java

A Linguagem de programação Java é uma linguagem de programação orientada a objetos, robusta, elegante e com todas as características das linguagens modernas, podendo ser utilizada nos mais diversos ambientes, desde aplicações simples no desktop até complexos aplicativos web, programação de robôs, redes de sensores, celulares e televisão digital interativa, além de muitos outros.

O JAVA se tornou a linguagem preferida para implementar aplicativos baseados na Internet e softwares para dispositivos que se comunicam por uma rede. Programas Java consistem em partes chamadas classes. As classes possuem partes chamadas métodos que realizam tarefas e retornam informações quando as tarefas são concluídas. È pssível entretanto utilizar classes existentes nas bibliotecas de classe Java que também são conhecidas como Java APIs (Application Programming Interfaces)

Arquitetura e Desenvolvimento

- Linguagem de programação orientada a objetos
- Familiar (sintaxe parecida com C)
- Simples e robusta (minimiza bugs, aumenta produtividade)
- Suporte nativo a threads (+ simples, maior portabilidade)
- Dinâmica (módulos, acoplamento em tempo de execução)
- Com coleta de lixo (menos bugs, mais produtividade)
- Independente de plataforma
- Segura (vários mecanismos para controlar segurança)
- Código intermediário de máquina virtual interpretado (compilação rápida - + produtividade no desenvolvimento)
- Sintaxe uniforme, rigorosa quanto a tipos (código mais simples, menos diferenças em funcionalidades iguais)

Linguagens como Java são linguagens **orientadas a objeto**. A programação nessa linguagem, chamada **programação orientada a objetos (*Object-Oriented Programming* — OOP)**, permite-lhe implementar um design orientado a objetos como um sistema funcional. Linguagens como o C, por outro lado, são **procedurais**, então a programação tende a ser **orientada para a ação**. No C, a unidade de programação é a **função**. Grupos de ações que realizam alguma tarefa comum são reunidos em funções e as funções são agrupadas para formar programas. No Java, a unidade de programação é a classe a partir da qual os objetos por fim são **instanciados** (criados). Classes Java contêm **métodos** (que implementam operações e são semelhantes a funções na linguagem C) bem como **campos** (que implementam atributos).

Programadores em Java concentram-se na criação de classes. Cada classe contém campos e o conjunto de métodos que manipulam os campos e fornecem serviços aos **clientes** (isto é, outras classes que utilizam a classe). O programador utiliza classes existentes como blocos de construção para construir novas classes.

As classes estão para os objetos assim como as plantas arquitetônicas estão para as casas. Assim como podemos construir muitas casas a partir de uma planta, podemos instanciar (criar) muitos objetos a partir de uma classe. Você não pode fazer refeições na cozinha de uma planta; isso só é possível em uma cozinha real.

As classes podem ter relacionamentos com outras classes. Por exemplo, em um design orientado a objetos de um banco, a classe "caixa de banco" precisa se relacionar com a classe "cliente", a classe "gaveta de dinheiro", a classe "cofre" etc. Esses relacionamentos são chamados de **associações**.

Empacotar software como classes possibilita que os sistemas de software futuros **reutilizem** as classes. Grupos de classes relacionadas são frequentemente empacotados como **componentes** reutilizáveis. Assim como corretores de imóveis costumam dizer que os três fatores mais importantes que afetam o preço dos imóveis são "localização, localização e localização", as pessoas na comunidade de software costumam dizer que os três fatores mais importantes que afetam o futuro do desenvolvimento de software são "reutilização, reutilização e reutilização". A reutilização de classes existentes ao construir novas classes e programas economiza tempo e esforço. A reutilização também ajuda-lhe a construir sistemas mais confiáveis e eficientes, porque classes e componentes existentes costumam passar por extensos testes, depuração e ajuste de desempenho.

Com a tecnologia de objetos, você pode construir grande parte do software necessário combinando classes, exatamente como fabricantes de automóveis combinam partes intercambiáveis. Cada nova classe que você criar terá o potencial de se tornar um "ativo de software" que você e outros programadores podem utilizar para acelerar e aprimorar a qualidade de seus esforços futuros no desenvolvimento de software.

Plataforma e Tipos de Aplicação

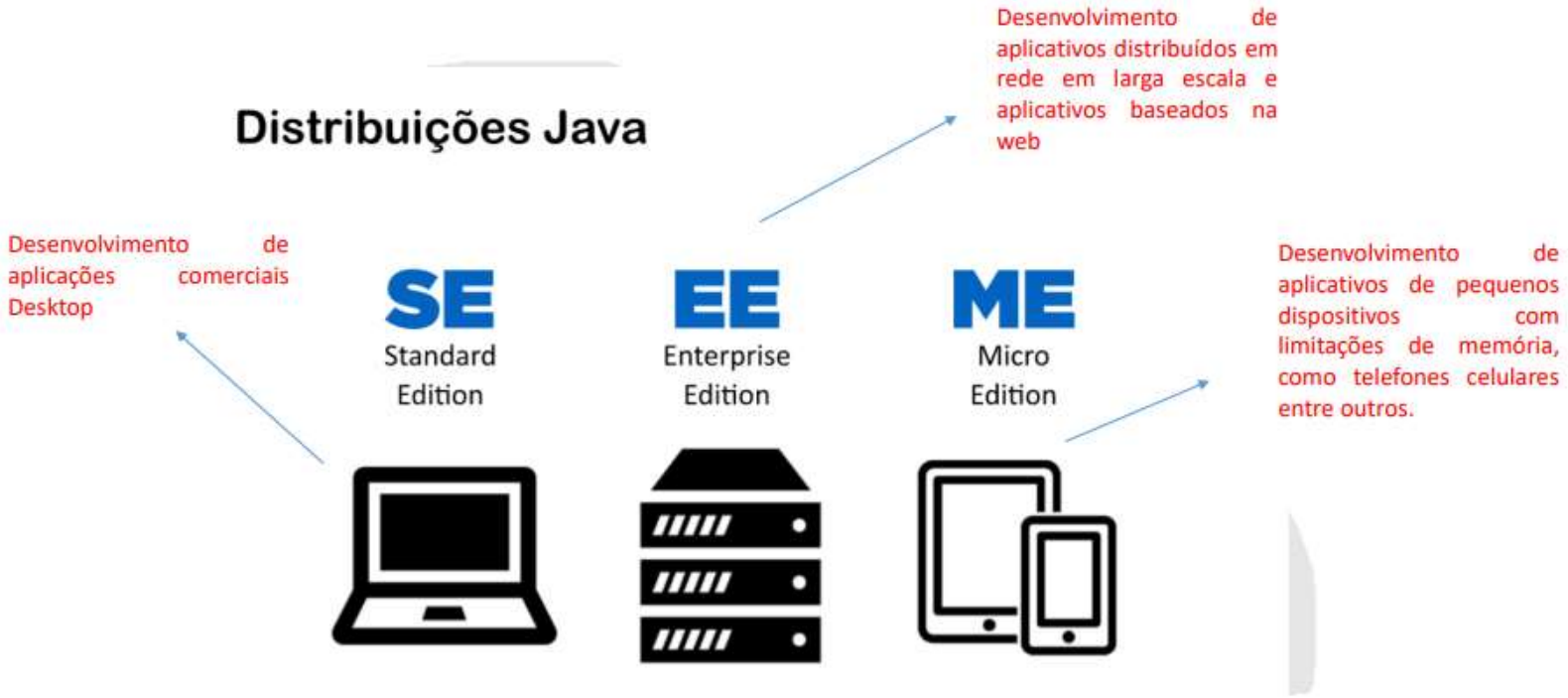
Plataformas: Java é portável, portanto pode ser executado em qualquer plataforma que tenha suporte a sua JVM.

Aplicativos: programas escritos em Java que não necessitam de um navegador para serem utilizados.

Applets: programas escritos em Java, normalmente carregados na WWW e executados por um navegador.

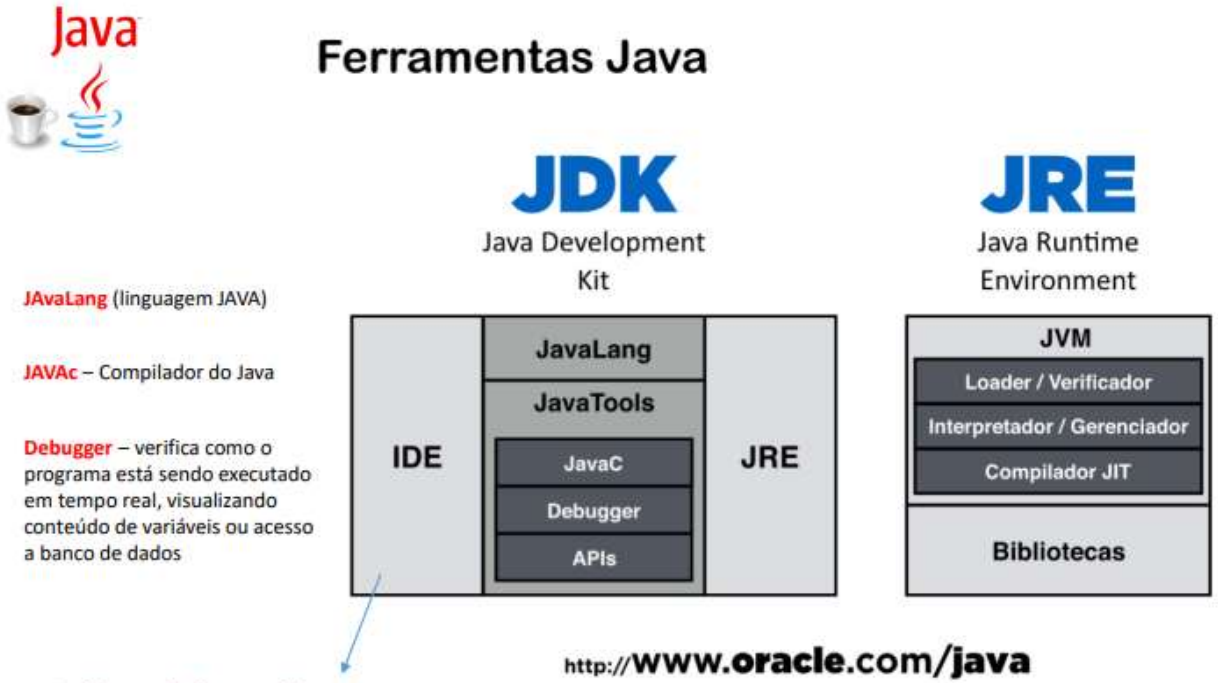
Servlets: programas escritos em Java, normalmente carregados na WWW e executados pelo Servidor WEB.

JAVA – EAGS SIN 2022



JAVA – EAGS SIN 2022

Ferramentas Java



JAvaLang (linguagem JAVA)

JAVAc – Compilador do Java

Debugger – verifica como o programa está sendo executado em tempo real, visualizando conteúdo de variáveis ou acesso a banco de dados

Ambiente de desenvolvimento. (Eclipse e Netbeans)

http://www.oracle.com/java

Loader - responsável por carregar o bytecode na sua memória da máquina virtual

Verificador verifica se este código pode ser executado ou não

Interpretador que irá transformar o Bytecode em código nativo da máquina que irá rodar

Gerenciador vai tratar como os códigos e as variáveis vão ser gerenciadas na memória da JVM

Compilador JIT – Just in time – conhecido também como compilador Hotspot compila a partes dos programas mais importantes e já compila definitivamente para dentro do bytecode) e Bibliotecas – APIS para poder utilizar dentro do seu programa)

JAVA – EAGS SIN 2022

Java é...

- **Livre** e **grátis**
- Popular e com **comunidade** ativa
- Altamente **extensível**
- **Compilado** para **bytecode**
- Executado por uma Máquina Virtual (**JVM**)
- Funcional em **múltiplas plataformas**
- Compatível com várias **bibliotecas**
- 100% **Orientado a Objetos**
- Compatível com **Coletor de Lixo** (Garbage Collector)
- Compatível com **Multithreading**

CONCEITOS BÁSICOS

- **A Garbage Collection** é a grande responsável pela liberação automática do espaço em memória. Isso acontece automaticamente durante o tempo de vida do programa Java.
- O comando **java** do JDK executa um aplicativo JAVA
- **Javadoc** é um gerador de documentação criado pela Sun Microsystems para documentar a API dos programas em Java, a partir do código-fonte. O resultado é expresso em HTML. É constituído, basicamente, por algumas marcações muitos simples inseridas nos comentários do programa.
- O Comando **JAVAC** compila um programa JAVA
- Um arquivo de programa java deve terminar o arquivo com a extensão **.java**
- Quando um programa Java é compilado, o arquivo produzido pelo compilador termina com a extensão de arquivo **.class**.
- O arquivo produzido pelo compilador Java contém **bytecodes** que são executados pela JVM.
- **JVM** – Java Virtual Machine – Máquina Virtual do JAVA
- OBS: ***JAVA é case Sensitive;***

JAVA – EAGS SIN 2022

Pacote JAVA.lang – pacote que vem com coisas simples, como escrever na tela, fazer cáculos aritméticos entre outras coisas básicas. NÂO precisa chamar pe[redacted] IMPORT;

Alguns pacotes – Java.Applet – Para criar aplicativos;

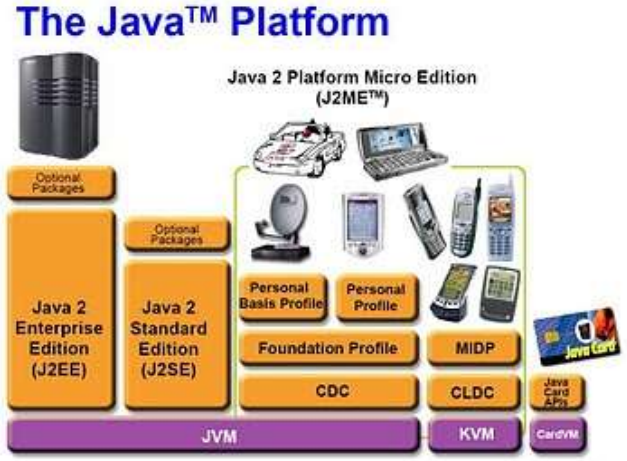
Java.util – para utilitários como monitoramento de teclado;
Java.util.date – possui métodos de manipulação de datas;
Java.util.stack – possui métodos de manipulação de pilhas;
Java.math – para funções matemáticas;
Java.net para redes;
Java.sound – bibliotecas estendidas para som/midea;
Java.awt – bibliotecas gráficas para janelas do JAVA;
Javax.swing – biblioteca que permite que se crie aplicações gráficas.

- Biblioteca AWT (abstract Window Toolkit – primeira biblioteca gráfica para janelas, deixa livre para que o SO escolhesse a aparência dos componentes;
- Biblioteca Swing – permite criar interfaces gráficas para janelas que pode ser utilizado em qualquer sistema operacional;

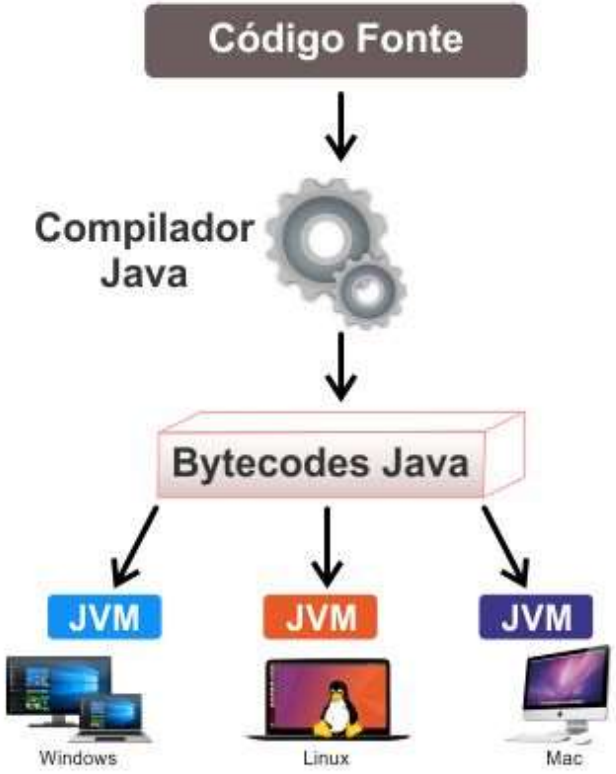
Javafx.fxml – Biblioteca Gráfica com possibilidade de criar telas gráficas que se adequa a qualquer dispositivo.
FXML (HTML+xml+CSS)

PLATAFORMAS

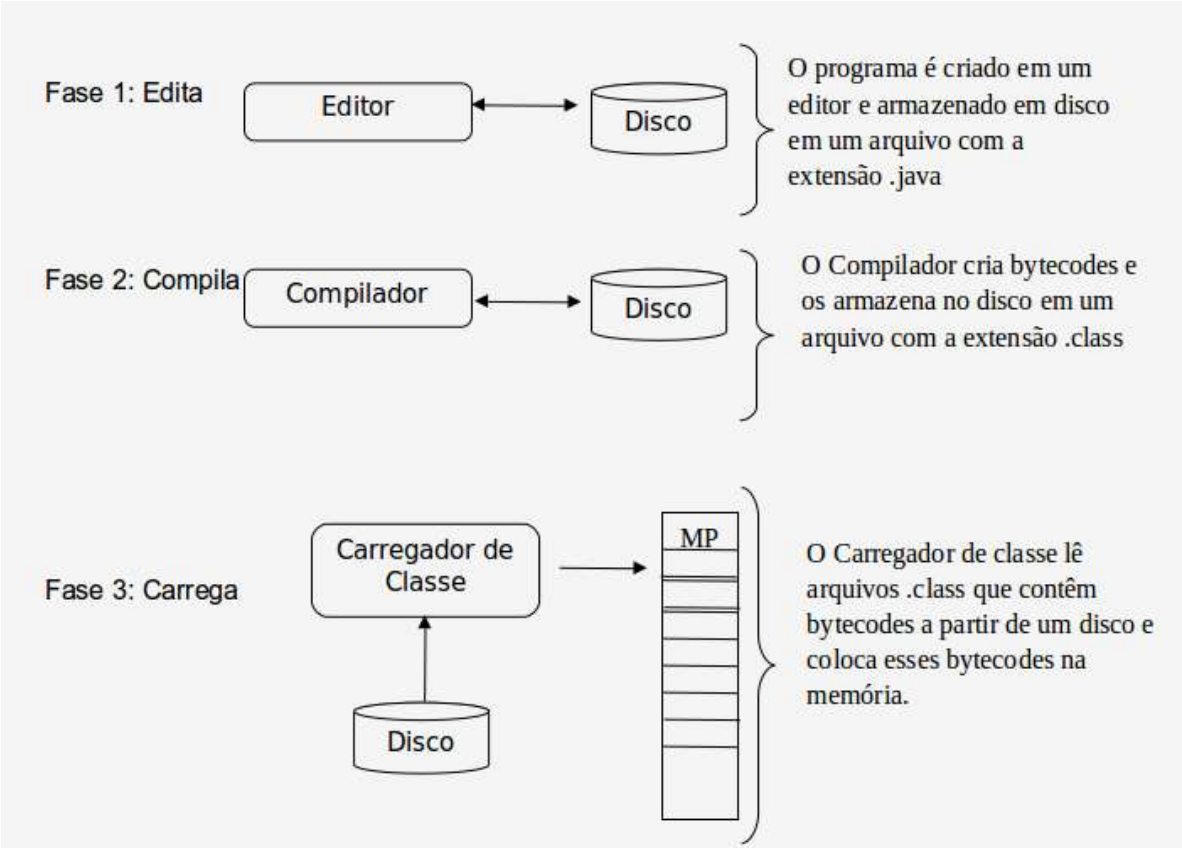
	Standard Edition	Enterprise Edition	Micro Edition
Versões em Março/2010	Java2SE	Java2EE	Java2ME
Caracteristicas	Console, desktop, applets, etc	Servlet, JSP, EJB, Componentes, Sistemas Distribuídos, etc	Dispositivos móveis, TV Digital, etc..



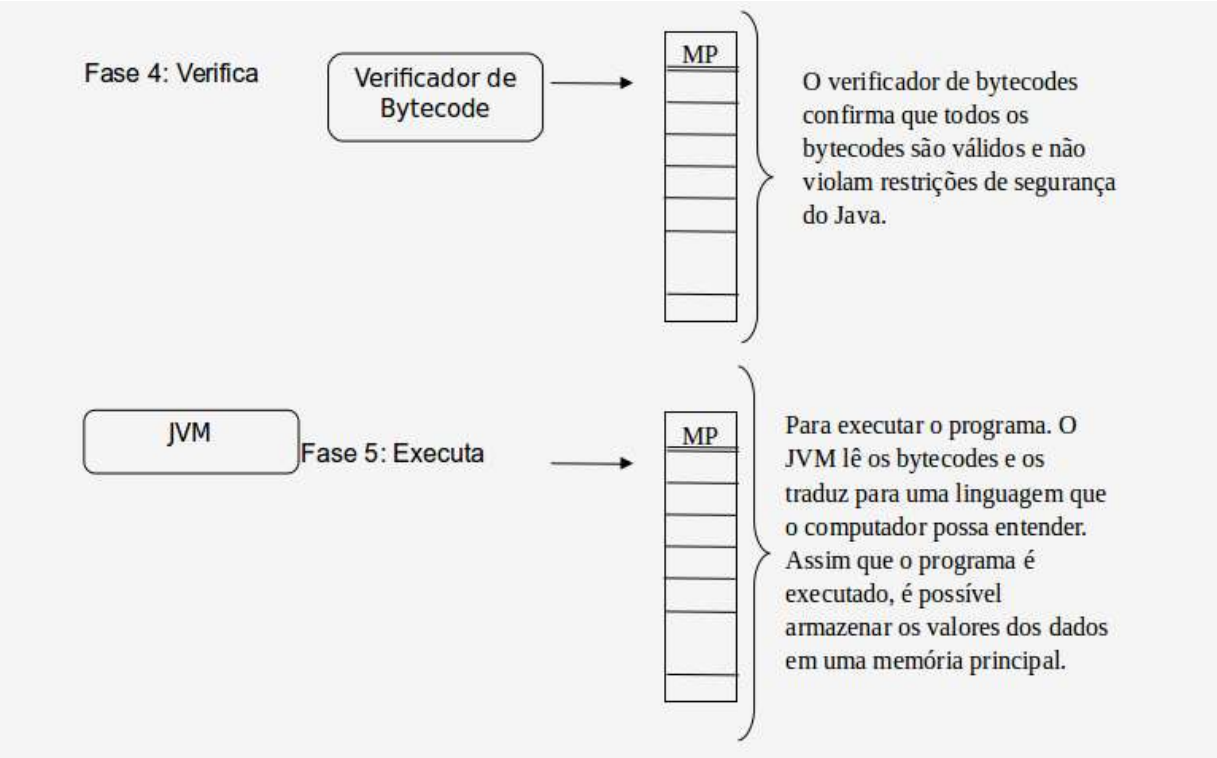
Ambiente de Desenvolvimento Java



Edição, compilação e execução



Edição, compilação e execução



JAVA – EAGS SIN 2022

```
public class PrimeiroExemplo {
    /*
     * Aí vai o meu primeiro
     * programa feito em Java!
     * Quanta emoção!
     */
    public static void main(String[] args) {
        System.out.println("Olá, Mundo!");
    } // Fim do método main()
} // Fim da classe
```

Nome da Classe

Nome do Método

Package primeiroprograma (OPCIONAL)
Public class PrimeiroPrograma {
Public static viod main(String[] args)

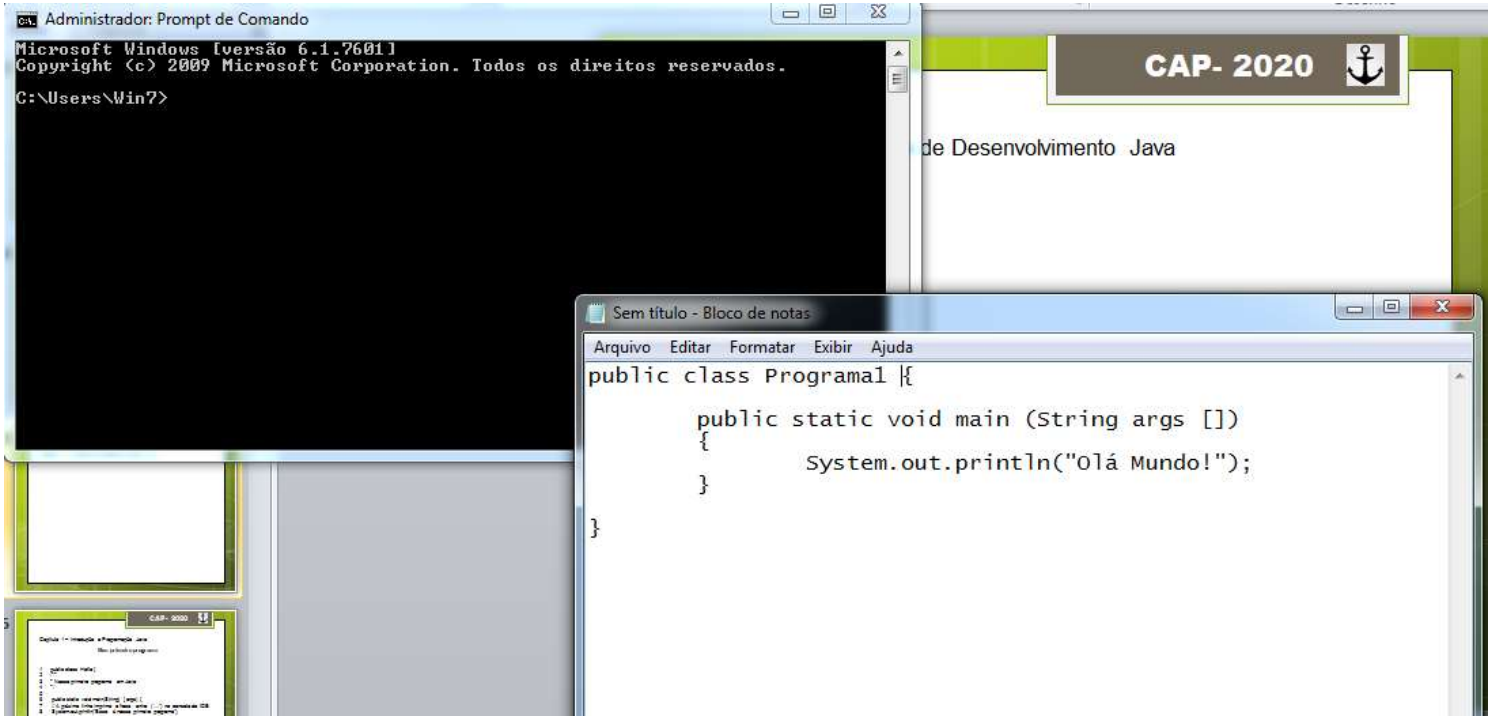
O método Main é o ponto de partida de cada aplicativo java e deve iniciar com public static void main (String [] args)

Meu primeiro programa

```
1    public class Hello {
2    /**
3    * Nosso primeiro programa em Java
4    */
5
6     public static void main(String[ ] args) {
7     // A próxima linha imprime a frase   entre (“…”) no console da IDE
8     System.out.println("Esse é nosso primeiro programa");
9       }
10  }
```

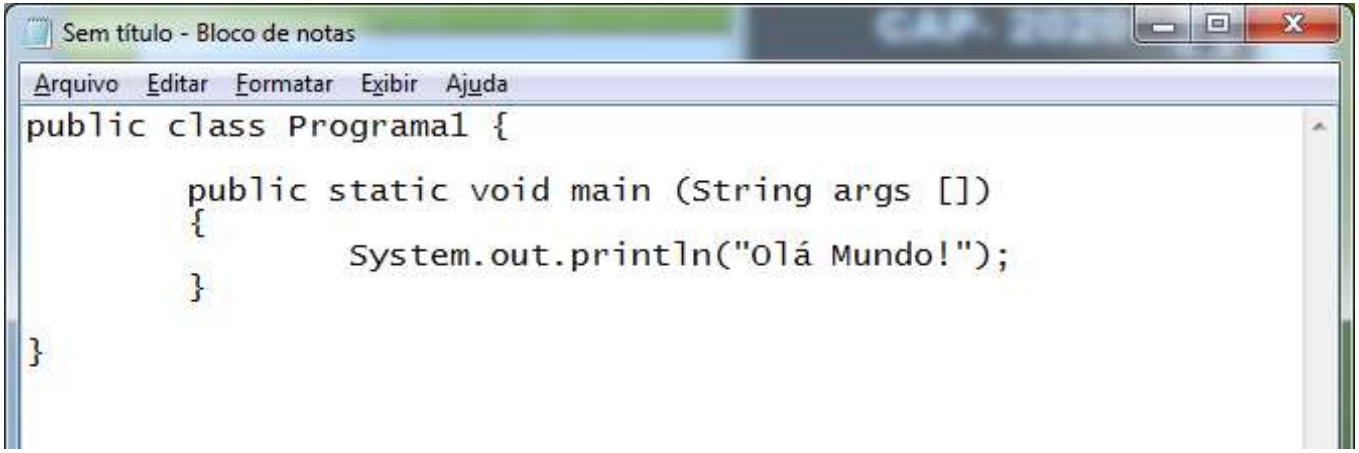
JAVA – EAGS SIN 2022

Ambiente de Desenvolvimento Java

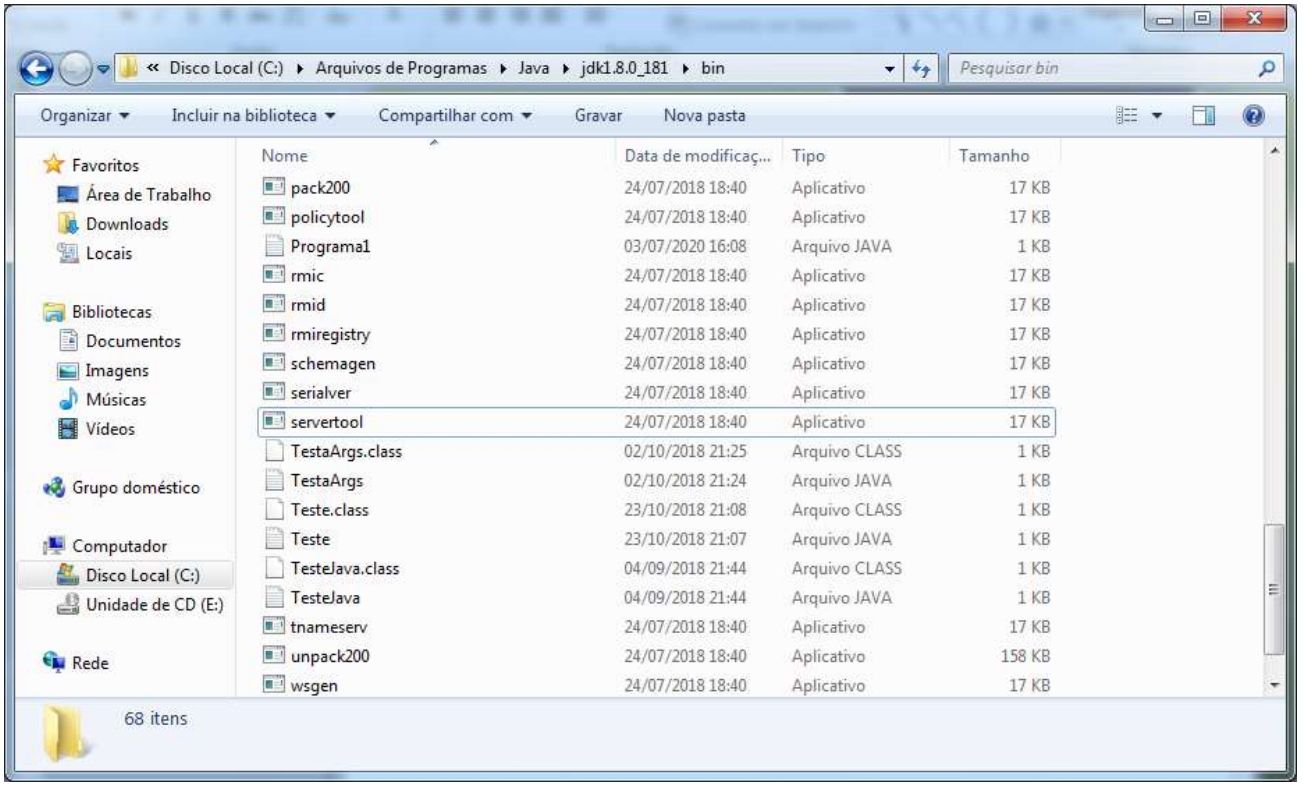


Salvar como Programa1.java em

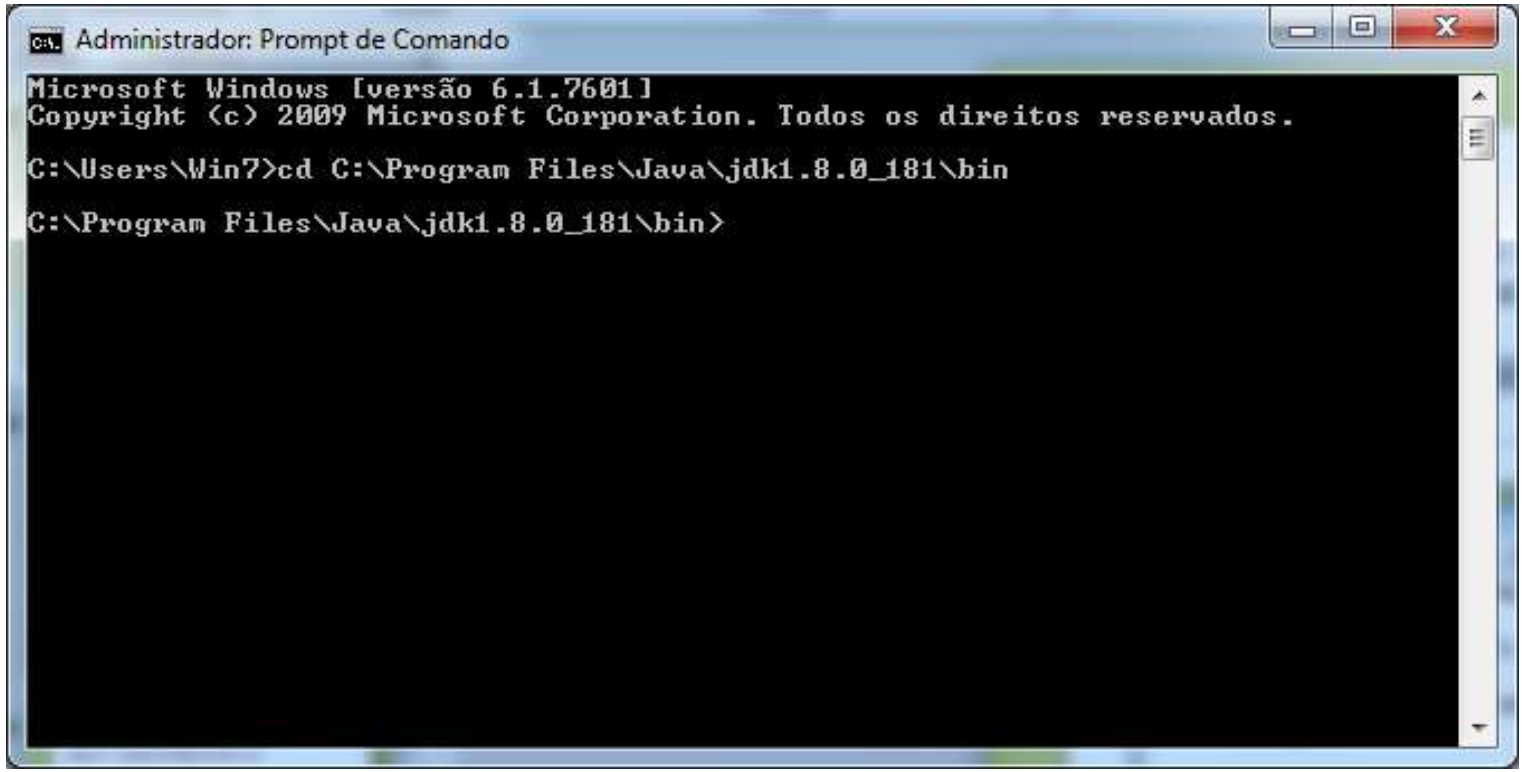
C:\Program Files\Java\jdk1.8.0_181\bin



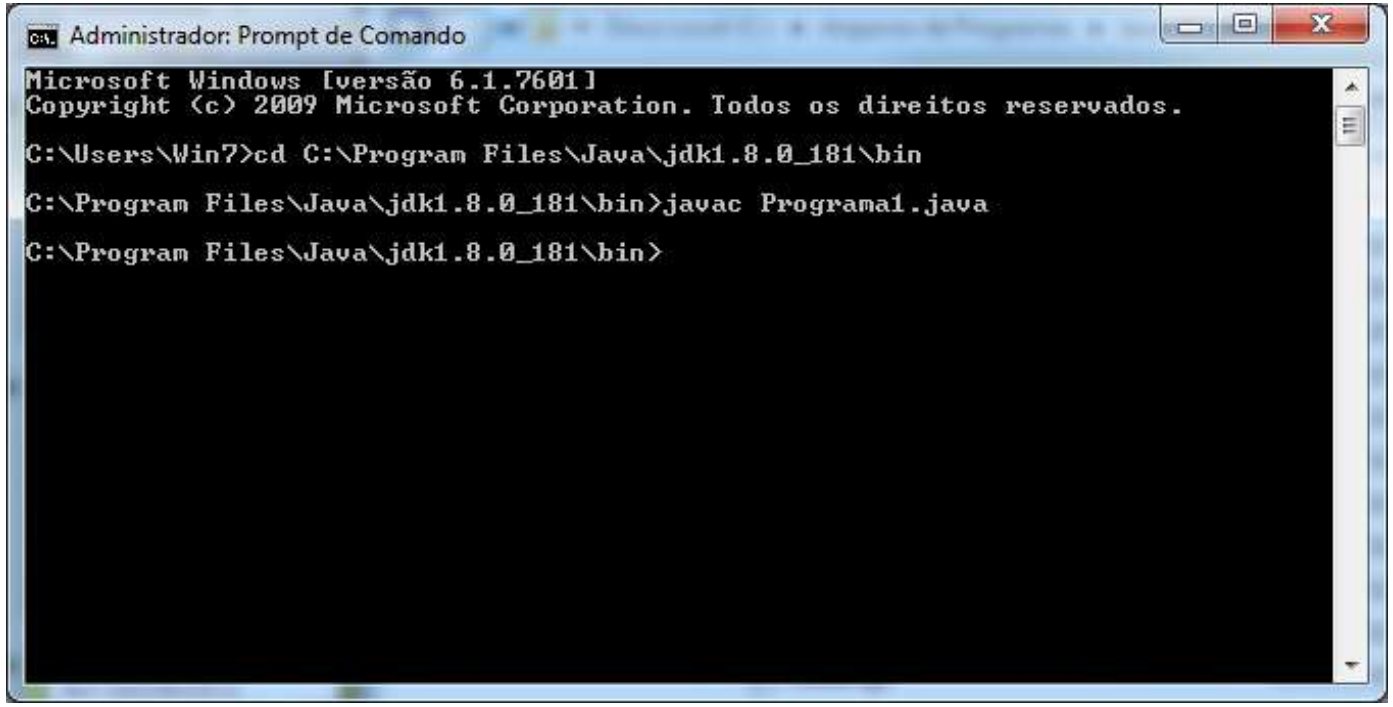
JAVA – EAGS SIN 2022



JAVA – EAGS SIN 2022



JAVA – EAGS SIN 2022



Modificando Programas

Uso do printf:

O método System.out.printf (f significa "formatted) exibe os dados formatados.

System.out.printf("%s\n%s\n", "Bem Vindo", "Ao Java");

Sequência de escape	Descrição
\n	Nova linha. Posiciona o cursor de tela no início da próxima linha.
\t	Tabulação horizontal. Move o cursor de tela para a próxima parada de tabulação.
\r	Retorno de carro. Posiciona o cursor da tela no início da linha atual — não avança para a próxima linha. Qualquer saída de caracteres depois do retorno de carro sobrescreve a saída de caracteres anteriormente gerados na linha atual.
\\	Barras invertidas. Utilizada para imprimir um caractere de barra invertida.

\"	Aspas duplas. Utilizada para imprimir um caractere de aspas duplas. Por exemplo,
	`System.out.println( "\"in quotes\"" );`
	exibe
	`"in quotes"`

Palavras-Chave Reservadas

Usadas para identificar tipos, modificadores e mecanismo de controle de fluxo.

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

* *Palavras reservadas mas não usadas*

Operadores do Java

Operação Java	Operador	Expressão algébrica	Expressão Java
Adição	+	$f + 7$	`f + 7`
Subtração	-	$p - c$	`p - c`
Multiplicação	*	bm	`b * m`
Divisão	/	x/y ou $\frac{x}{y}$ ou $x \div y$	`x / y`
Resto	%	$r \bmod s$	`r % s`

Figura 2.11 | Operadores aritméticos.

Operador	Operação	Ordem de avaliação (precedência)
* / %	Multiplicação Divisão Resto	Avaliado primeiro. Se houver vários operadores desse tipo, eles são avaliados da esquerda para a direita.
+ -	Adição Subtração	Avaliado em seguida. Se houver vários operadores desse tipo, eles são avaliados da esquerda para a direita.
=	Atribuição	Avaliado por último.

Figura 2.12 | Precedência de operadores aritméticos.

Operadores do Java

Operador de igualdade ou relacional algébrico padrão	Operador de igualdade ou relacional Java	Exemplo de condição em Java	Significado da condição em Java
Operadores de igualdade			
=	==	x == y	x é igual a y
≠	!=	x != y	x é diferente de y
Operadores relacionais			
		y	x é maior que y
<	<	x < y	x é menor que y
≥	>=	x >= y	x é maior que ou igual a y
≤	<=	x <= y	x é menor que ou igual a y

Figura 2.14 | Operadores de igualdade e operadores relacionais.

Operadores	Associatividade	Tipo
* / %	da esquerda para a direita	multiplicativo
+ -	da esquerda para a direita	aditivo
=	da esquerda para a direita	relacional
== !=	da esquerda para a direita	igualdade
=	da direita para a esquerda	atribuição

Figura 2.16 | Operadores de precedência e de associatividade discutidos.

IDENTIFICADORES

Identificadores são os nomes que são associados às estruturas de um programa em Java, seja o nome de uma classe, variável ou método.
Vale lembrar que o Java é case-sensitive, ou seja, diferencia caracteres maiúsculos de minúsculos. Assim, um recurso do código cujo identificador é Teste é diferente de outro recurso cujo identificador é teste.

MinhaClasse
meuAtributo
minhaVariável
meuMétodo
meu_pacote
MINHA_CONSTANTE

Existem inclusive regras de boas práticas a serem seguidas nas definições de letras maiúsculas e minúsculas dos identificadores de acordo com o Code Conventions da Sun (http://java.sun.com/docs/codeconv).

Basicamente, classes se iniciam com letras maiúsculas e o restante das letras minúsculas; se o identificador de classe for composto de mais de uma palavra, elas virão aglutinadas, cada uma delas também se inicia com maiúsculas. Como em: ExemploDeNomeDeClasse. Para método e variável (ou atributo) vale a mesma regra, sendo que a primeira palavra se inicia com letra minúscula. Exemplo:

exemploDeNomeDeMetodo( )
ExemploDeNomeDeVariavel

Existem algumas outras regras, por isso recomendamos a leitura do Code Conventions. Quanto ao universo que pode ser utilizado, os identificadores podem começar com qualquer letra, underscore (_) ou cifrão ($). Os caracteres subsequentes podem utilizar também números de 0 a 9.

Lista de operadores do Java

OPERADOR	FUNÇÃO	OPERADOR	FUNÇÃO
+	Adição	~	Complemento
-	Subtração	<<	Deslocamento à esquerda
*	Multiplicação	>>	Deslocamento à direita
/	Divisão	>>>	Desloc. a direita com zeros
%	Resto	=	Atribuição
++	Incremento	+=	Atribuição com adição
--	Decremento	-=	Atribuição com subtração
>	Maior que	*=	Atribuição com multiplicação
>=	Maior ou igual	/=	Atribuição com divisão
<	Menor que	%=	Atribuição com resto
<=	Menor ou igual	&=	Atribuição com AND
==	Igual	|=	Atribuição com OR
!=	Não igual	^=	Atribuição com XOR
!	NÃO lógico	<<=	Atribuição com desl. esquerdo
&&	E lógico	>>=	Atribuição com desloc. direito
||	OU lógico	>>>=	Atrib. C/ desloc. a dir. c/ zeros
&	AND	? :	Operador ternário
^	XOR	(*tipo*)	Conversão de tipos (cast)
|	OR	instanceof	Comparação de tipos

VARIÁVEIS

→ **Definição**

Variáveis são espaços de memória utilizados para guardar dados de um determinado tipo.

→ **Nomenclatura**

O nome de uma variável pode ser qualquer identificador válido. Identificadores válidos podem conter somente:
Letras (‘a’ a ‘z’ e ‘A’ a ‘Z’)
Dígitos (0 a 9)
Sublinhados (_)
Sinais de Cifrão ($)

Cuidado!!!!
INICIAR um identificador com dígito(s) causa erro de compilação!

Classificação das variáveis

Variáveis podem receber até 4 classificações (ao mesmo tempo) de acordo com a sua visibilidade, escopo, constância e tipo de dado. Objetivamente, visibilidade está relacionado ao uso de modificadores de acesso; escopo ao contexto em que a variável foi definida; constância ao fato de uma variável ser ou não uma constante; e tipo de dado ao tipo de informação que a variável é capaz de armazenar.

Visibilidade (04)

- public
- private
- protected
- padrão ou default

Escopo (03)

- **classe (static)**

Atributos declarados com o modificador static. São acessíveis diretamente pela classe, sem a necessidade de instanciar um objeto. Espécie de variável "global" compartilhada entre os objetos da classe.

- **instância**

Atributos declarados sem o modificador static. Cada instância de uma classe possui o seu próprio conjunto de variáveis de instância.

- **local**

Variáveis declaradas dentro de um método ou bloco de código. São variáveis temporárias, válidas apenas dentro do método ou bloco onde foi declarada.

Classificação das variáveis

Mutabilidade (02)

- **constante (final)**

São variáveis não modificáveis após a sua inicialização, isto é, são constantes. Utiliza-se a palavra-chave final antes do tipo de dado da variável para especificar que ela é uma constante. Constantes podem ser inicializadas diretamente no momento da declaração ou por meio de construtores. Quando é inicializada diretamente na declaração, significa que o valor da constante será o mesmo para todos os objetos da classe.

- **não constante**

O valor da variável pode ser modificado a qualquer momento.

Tipo de dado (02)

- **primitiva**

Quando o tipo de dados da variável é um tipo primitivo. Guarda um valor compatível com o tipo primitivo declarado.

- **de referência**

Variável de referência ou de instância: quando o tipo de dados da variável é uma classe. Guarda uma referência para um objeto da classe declarada.

Acesso a variável da Classe

```
1     public class Teste1 {
2
3     int x=0;
4
5
6      public static void main(String[ ] args) {
7
8        System.out.println(x);
9          }
10  }
```

Acesso a variável da Classe – usando STATIC

```
1     public class Teste1 {
2
3     static int x=0;
4
5
6      public static void main(String[ ] args) {
7
8      System.out.println(x);
9        }
10  }
```

Acesso a variável da Classe – usando INSTÂNCIA

```
1     public class Teste1 {
2
3     int x=0;
4
5
6      public static void main(String[ ] args) {
7      Teste1 teste1=new Teste1();
8      System.out.println(teste1.x);
9        }
10  }
```

Inicialização de variáveis

```
1     public class Teste1 {
2
3     static int x;
4
6      public static void main(String[ ] args) {
7       int y;
8    System.out.println(y);
9    System.out.println(x);
10         }
11  }
```

Tipos de Dados Primitivos

Tipo	Valores	Tamanho em bits	Padrão
boolean	true ou false		
char	‘\u0000’ a ‘\uffff’ (0 a 65535)	16	(conjunto de caracteres unicode)
byte	-128 a +127	8	
short	-32768 a +32767	16	
int	-2147483648 a +2147483647	32	
long	-2 ^63 a 2^63-1	64	
float	1.4e ^-45 a 3.4e+^38	32	Ponto flutuante IEEE 754
double	5e-^324 a 1.8e+^38	64	Ponto flutuante IEEE 754

Exercícios

1.2 Preencha as lacunas em cada uma das seguintes frases sobre o ambiente Java:

a) O comando _____ do JDK executa um aplicativo Java.

b) O comando _____ do JDK compila um programa Java.

c) Um arquivo de programa Java deve terminar com a extensão de arquivo _____.

d) Quando um programa Java é compilado, o arquivo produzido pelo compilador termina com a extensão de arquivo _____.

e) O arquivo produzido pelo compilador Java contém _____ que são executados pela Java Virtual Machine.

JAVA – EAGS SIN 2022

Questão 39 – CAP 2009 – Prova Amarela

39) Em relação à linguagem de programação Java, complete corretamente as lacunas das sentenças abaixo, e assinale a opção corrtea.

I - O comando _____ do J2SE Development Kit executa um aplicativo Java.
II - O comando _____ do J2SE Development Kit compila um programa Java.
III- Um arquivo de programa Java deve terminar com a extensão de arquivo _____.
IV - Quando um programa Java é compilado, o arquivo produzido pelo compilador termina com a extensão de arquivo _____.
V - O arquivo produzido pelo compilador Java contém _____ que são executados pela Java Virtual Machine.

(A) java / javac / java / class / bytecodes
(B) javac / java / class / java / bytecodes
(C) main / javac / java / jar / class
(D) java / class / rar / jar / javac
(E) rar / jar / java / javac / class

Questão 7 – CAP 2010 – Prova Amarela

7) Analise as linhas de código a seguir escritas em linguagem Java.

```
1. a++;
2. if (a=c && a > b) System.out.println("teste");
3. b = a ** b;
4. c = a%b;
5. a = a-b:
```

Dentre as linhas de código acima, quais apresentam apenas expressões sintaticamente corretas, isto, que seriam compiladas sem erro (considere as variáveis a, b e c previamente declaradas como do tipo **double**)?

(A) 1, 2, 3
(B) 1, 2, 5
(C) 2, 3, 4
(D) 1, 4
(E) 1, 2, 4

Questão 9 – CAP 2010 – Prova Amarela

9) Assinale a opção que contém somente tipos primitivos da linguagem Java:

(A) boolean, string ,char, integer, long, float, double
(B) boolean, bit, char, short, integer, long, Float, Double
(C) Boolean, byte, string, short, int, long, float, Double
(D) boolean, string ,char, short, int, Long, float, double
(E) boolean, byte, char, short, int, long, float, double

Questão 39 – CAP 2011 – Prova Amarela

39) Em JAVA, segundo Robert W. Sebesta (2002), que palavra reservada é utilizada para especificar em uma definição de classe que a referida classe não pode ser pai de nenhuma subclasse?

(A) final
(B) Extends
(C) Implements
(D) Class
(E) Public

Exercícios

1) Em linguagem Java:

a) == significa atribuição. & significa “E” lógico. || significa “OU” lógico.
b) == significa igualdade. && significa atribuição lógica. || significa “+” lógico.
c) == significa igualdade. && significa “E” lógico. || significa “OU” lógico.
d) <> significa igualdade. &+ significa “E” lógico. | significa “OU” lógico.
e) =+ significa igualdade superior. && significa “E” lógico. |=| significa “OU” lógico.

2) Para declarar uma constante chamada SIZE do tipo inteiro e com valor 25 no JAVA, pode-se utilizar a construção

A) this extend integer SIZE := 25;
B) this final int SIZE := 25;
C) static final int SIZE=25;
D) void final int SIZE =25;
E) static extend integer SIZE = 25;

CAP 2012- Amarela

3) Considerando o tratamento de parâmetros na linguagem JAVA, analise o programa abaixo, desenvolvido no ambiente NetBeans 7.2.1:

```
package p5;
public class P5 { // Netbeans IDE 7.2.1

    public static void main(String[] args) {
        int c;
        c = 5;

        System.out.println(c);
        System.out.println(c++);
        System.out.println(c);

        System.out.println(c--);
        System.out.println(++c);
        System.out.println(c);
    }
}
```

Dentre as opções abaixo, qual apresenta o valor que será impresso por esse programa ao final de sua execução?

(A) 5 5 5 4 5 5
(B) 5 5 6 5 6 6
(C) 5 5 6 6 6 6
(D) 5 6 5 4 5 5
(E) 5 6 5 5 5 5

CAP 2012- Amarela

4) Considerando os valores e os tipos de dados em JAVA, analise o programa abaixo, desenvolvido no ambiente NetBeans 7.2.1:

```
package cap6;
public class CAP6 {

     public static void main(String[] args) {

     final int i=1;
     System.out.print(i++);

    }
}
```

Assinale a opção correta com relação a execução desse programa.

(A) O programa, ao ser executado, entra em LOOP
(B) Não é possível alterar o valor de variável tipo final
(C) Imprime o número 2
(D) Imprime o número 1
(E) Imprime o número 0

CAP 2011- Amarela

48) Analise o código em JAVA a seguir.

```
package prova;
public class Main {

    public static void main(String[] args) {
        int b = 0;
        char a = 25;
        b = (int) a;
        System.out.println(b);
    }
}
```

Sabendo-se que o código acima foi escrito e executado utilizando o IDE NetBeans 6.0.1, assinale a opção correta referente ao valor da variável b que será impresso.

(A) 21
(B) 22
(C) 23
(D) 24
(E) 25

17) No contexto da Linguagem Java, em qual das opções abaixo todos os identificadores (separados por vírgula) são válidos?

(A) *soma, ___nome, $salario, :resultado*
(B) *_$aumento, nomeDoFuncionarioDoMes, 45andar*
(C) *peso, ___nome_Do_Funcionario, $salario_Mensal, idade3*
(D) *.marca, 3vezes, $salario, :numero*
(E) *.classe, _nome1, public, IDADE*

ESTRUTURAS DE CONTROLE

Instruções de seleção em Java

O Java contém três tipos de **instruções de seleção** (discutidos neste capítulo e no Capítulo 5). A instrução `if` realiza uma ação (seleciona) se uma condição for verdadeira ou pula a ação se a condição for falsa. A instrução `if...else` realiza uma ação se uma condição for verdadeira e realiza uma ação diferente se a condição for falsa. A instrução de seleção `switch` (Capítulo 5) realiza uma de muitas ações diferentes, dependendo do valor de uma expressão.

A instrução `if` é uma **instrução de seleção única** porque seleciona ou ignora uma única ação (ou, como veremos a seguir, um único grupo de ações). A instrução `if...else` é chamada **instrução de seleção dupla** porque seleciona entre duas ações diferentes (ou grupos de ações). A instrução `switch` é chamada de **instrução de seleção múltipla** pois seleciona entre muitas ações diferentes (ou grupos de ações).

Instruções de repetição em Java

O Java fornece três **instruções de repetição** (também chamadas **instruções de loop**) que permitem que programas executem instruções repetidamente contanto que uma condição (chamada **condição de continuação do loop**) permaneça verdadeira. As instruções de repetição são as instruções `while`, `do...while` e `for`. (O Capítulo 5 apresenta as instruções `do...while` e `for`.) As instruções `while` e `for` realizam a ação (ou grupo de ações) no seu corpo zero ou mais vezes — se a condição de continuação de loop for inicialmente falsa, a ação (ou grupo de ações) não será executada. A instrução `do...while` realiza a ação (ou grupo de ações) no seu corpo uma ou mais vezes. As palavras `if`, `else`, `switch`, `while`, `do` e `for` são palavras-chave Java. Uma lista completa das palavras-chave Java é apresentada no Apêndice C.

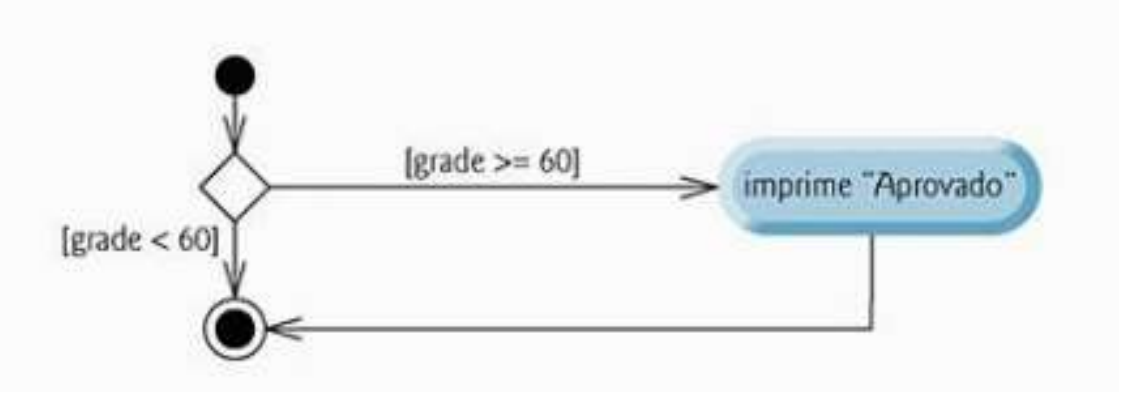
Resumo das instruções de controle em Java

O Java contém somente três tipos de estruturas de controle, que daqui para a frente chamaremos de instruções de controle: a instrução de sequência, instruções de seleção (três tipos) e instruções de repetição (três tipos). Cada programa é formado combinando a quantidade de instruções apropriada para o algoritmo que o programa implementa. Podemos modelar cada instrução de controle como um diagrama de atividade. Como na Figura 4.1, cada diagrama contém um estado inicial e um estado final que representa um ponto de entrada e um ponto de saída da instrução de controle, respectivamente. As **instruções de controle de entrada única/saída única** facilitam a construção de programas — basta conectarmos o ponto de saída de uma instrução ao ponto de entrada da instrução seguinte. Chamamos isso de **empilhamento de instruções de controle**. Aprenderemos que existe apenas outra maneira de conectar instruções de controle — **aninhamento de instruções de controle** — em que uma instrução de controle aparece dentro da outra. Portanto, algoritmos nos programas Java são construídos a partir de apenas três tipos de instruções de controle, combinadas apenas de duas maneiras. Isso é a essência da simplicidade.

A Instrução de seleção if

Se a nota do aluno for maior que ou igual a 60
Imprima "Aprovado"

```
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
```



A Instrução de seleção dupla if..else

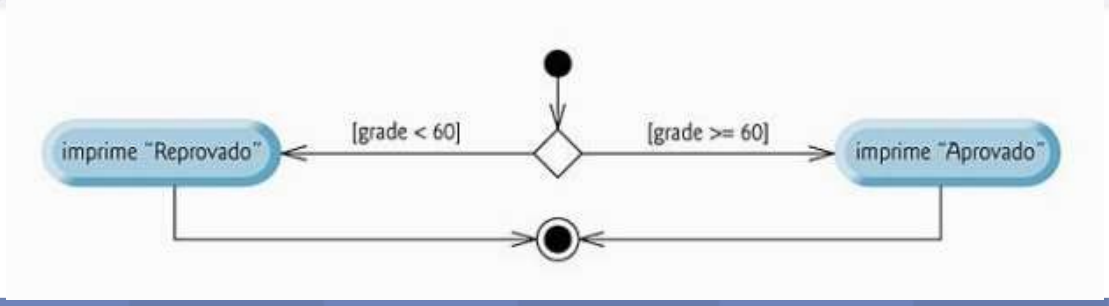
A instrução `if` de seleção única realiza uma ação indicada somente quando a condição é `true`; caso contrário, a ação é pulada. A **instrução de seleção dupla `if...else`** permite especificar uma ação a realizar quando a condição é verdadeira e uma ação diferente quando a condição é falsa. Por exemplo, este pseudocódigo:

```
Se a nota do aluno for maior que ou igual a 60
Imprima "Aprovado"
Caso contrário (Else)
Imprima "Reprovado"
```

imprime "Aprovado" se a nota do aluno for maior ou igual a 60, mas imprime "Reprovado" se for menor que 60. Em qualquer um dos casos, depois que impressão ocorre, a próxima instrução do pseudocódigo na sequência é "realizada".

A instrução *If...Else* no pseudocódigo anterior pode ser escrita em Java assim:

```
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
else
   System.out.println( "Failed" );
```



A Instrução if..else aninhadas

Um programa pode testar múltiplos casos colocando instruções `if...else` dentro de outras instruções `if...else` para criar **instruções `if...else` aninhadas**. Por exemplo, o pseudocódigo a seguir representa uma `if...else` aninhada que imprime `A` para notas de exame maiores que ou igual a 90, `B` para notas de 80 a 89, `C` para notas de 70 a 79, `D` para notas de 60 a 69 e `F` para todas as outras notas:

Se a nota do aluno for maior que ou igual a 90
    Imprima "A"
Caso contrário
    Se a nota do aluno for maior que ou igual a 80
        Imprima "B"
    Caso contrário
        Se a nota do aluno for maior que ou igual a 70
            Imprima "C"
        Caso contrário
            Se a nota do aluno for maior que ou igual a 60
                Imprima "D"
            Caso contrário
                Imprima "F"

Esse pseudocódigo pode ser escrito em Java como:

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else
   if ( studentGrade >= 80 )
      System.out.println( "B" );
   else
      if ( studentGrade >= 70 )
         System.out.println( "C" );
      else
         if ( studentGrade >= 60 )
            System.out.println( "D" );
         else
            System.out.println( "F" );
```

Operador Condicional

```
System.out.printIn ( studentGrade>=60 ? "Passed" : "Failed" );
```

Trabalhando com blocos

```
if ( grade >=  60 )
   System.out.println( "Passed" );
else
{
   System.out.println( "Failed" );
   System.out.println( "You must take this course again." );
}
```

A Instrução Switch .. Case

A declaração **switch** ainda possui um bloco definido como **default**, que é executado caso nenhum dos valores do **case** correspondam ao valor da variável inteira.

O trecho de código abaixo exemplifica uma implementação da declaração **switch:**

```
int andar = 3;
switch(andar){
    case 1:
        System.out.println("Você escolheu o 1º andar!");
        break;
    case 2:
        System.out.println("Você escolheu o 2º andar!");
        break;
    case 3:
        System.out.println("Você escolheu o 3º andar!");
        break;
    default:
        System.out.println("ESCOLHA UM ANDAR VÁLIDO!");
}
```

Operadores de atribuição Composta

Os operadores de atribuição composta abreviam expressões de atribuição. Instruções como:

variável = *variável* *operador* *expressão*;

onde *operador* é um dos operadores binários +, -, *, / ou % (ou outros que discutiremos mais adiante), pode ser escrita na forma:

variável *operador*= *expressão*;

Por exemplo, você pode abreviar a instrução:

```
c = c + 3;
```

com o **operador de atribuição composta de adição**, +=, como:

```
c += 3;
```

O operador += adiciona o valor da expressão na sua direita ao valor da variável na sua esquerda e armazena o resultado na variável no lado esquerdo do operador. Portanto, a expressão de atribuição `c += 3` adiciona 3 a `c`. A Figura 4.13 mostra os operadores aritméticos de atribuição composta, expressões de exemplo que utilizam os operadores e explicações do que os operadores fazem.

Operador de atribuição	Expressão de exemplo	Explicação	Atribuições
Suponha: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
+=	`c += 7`	`c = c + 7`	10 a c
-=	`d -= 4`	`d = d - 4`	1 a d
*=	`e *= 5`	`e = e * 5`	20 a e
/=	`f /= 3`	`f = f / 3`	2 a f
%=	`g %= 9`	`g = g % 9`	3 a g

Operadores de incremento e decremento

Operador	Nome do operador	Expressão de exemplo	Explicação
++	pré-incremento	++a	Incrementa a por 1, então utiliza o novo valor de a na expressão em que a reside.
++	pós-decremento	a++	Utiliza o valor atual de a na expressão em que a reside, então incrementa a por 1.
--	pré-incremento	--b	Decrementa b por 1, então utiliza o novo valor de b na expressão em que b reside.
--	pós-decremento	b--	Utiliza o valor atual de b na expressão em que b reside, então decrementa b por 1.

Precedências

Operadores						Associatividade	Tipo
++	--					da direita para a esquerda	unário pós-fixo
++	--	+	-	(*tipo*)		da direita para a esquerda	unário pré-fixo
*	/	%				da esquerda para a direita	multiplicativo
+	-					da esquerda para a direita	aditivo
=						da esquerda para a direita	relacional
==	!=					da esquerda para a direita	igualdade
?:						da direita para a esquerda	ternário condicional
=	+=	-=	*=	/=	%=	da direita para a esquerda	atribuição

ESTRUTURAS DE REPETIÇÃO

Estruturas de repetição são comandos JAVA que permitem repetir várias vezes um determinado bloco do programa até alguma condição de parada.

- While
- For
- Do..while

A instrução de repetição while

Uma **instrução de repetição** (ou um **loop**) permite especificar que um programa deve repetir uma ação enquanto alguma condição permanece verdadeira. A instrução de pseudocódigo

Enquanto houver mais itens em minha lista de compras
Compre o próximo item e risque-o da minha lista

descreve a repetição que ocorre durante um passeio de compras. A condição "enquanto houver mais itens em minha lista de compras" pode ser verdadeira ou falsa. Se ela for verdadeira, então a ação "Compre o próximo item e risque-o de minha lista" é realizada. Essa ação será realizada repetidamente enquanto a condição permanecer verdadeira. A(s) instrução(ões) contida(s) na instrução de repetição *While* constitui(em) seu corpo, que pode ser uma instrução única ou um bloco. Por fim, a condição se tornaria falsa (quando o último item na lista de compras foi comprado e riscado da lista). Nesse ponto, a repetição termina e a primeira instrução depois da instrução de repetição é executada.

Como exemplo da **instrução de repetição `while`** do Java, considere um segmento de programa projetado para encontrar a primeira potência de 3 maior que 100. Suponha que a variável `int product` tenha sido inicializada como 3. Quando a instrução `while` seguinte terminar a execução, `product` conterá o resultado:

```
while ( product <= 100 )
   product = 3 * product;
```

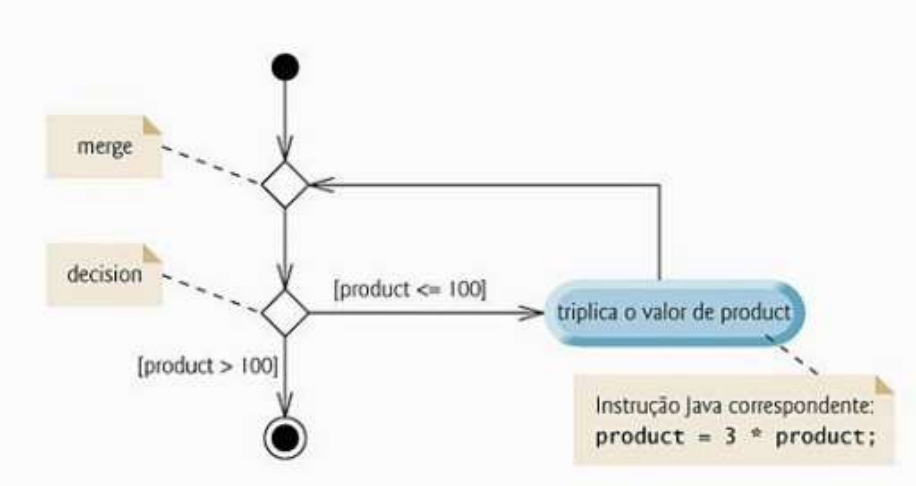
A instrução de repetição while

O **while** executa um bloco de comandos repetidamente, enquanto a condição lógica imposta como teste for verdadeira (expressão lógica igual a true).
A declaração **while** possui a seguinte sintaxe:

while (expressão_lógica) {
instrução1;
instrução2;
...
}

Exemplo:

```
int i = 10;
while (i > 0){
    System.out.println(i);
    i--;
}
```



Princípios básicos de repetição controlada por contador

1. Uma **variável de controle** (ou contador de loop).
2. O **valor inicial** da variável de controle.
3. O **incremento** (ou **decremento**) pelo qual a variável de controle é modificada a cada passagem pelo loop (também conhecido como **cada iteração do loop**).
4. A **condição de continuação do loop** que determina se o loop deve continuar.

Para ver esses elementos da repetição controlada por contador, considere o aplicativo da Figura 5.1, que utiliza um loop para exibir os números de 1 a 10.

```
1  // Figura 5.1: WhileCounter.java
2  // Repetição controlada por contador com a instrução de repetição while.
3
4  public class WhileCounter
5  {
6     public static void main( String[] args )
7     {
8        int counter = 1; // declara e inicializa a variável de controle
9
10       while (counter <= 10 ) // condição de continuação do loop
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter; // incrementa a variável de controle por 1
14       } // fim do while
15
16       System.out.println(); // imprime uma nova linha
17    } // fim de main
18 } // fim da classe WhileCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

Instrução de repetição for

O **for** executa um bloco de comandos repetidamente da mesma forma que o **while** ou o **do-while**, ou seja, o programador pode controlar a quantidade de execuções de acordo com um teste lógico.
A declaração for possui a seguinte sintaxe:

for (expressão_lógica) {
instrução1;
instrução2;
...
}

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

=

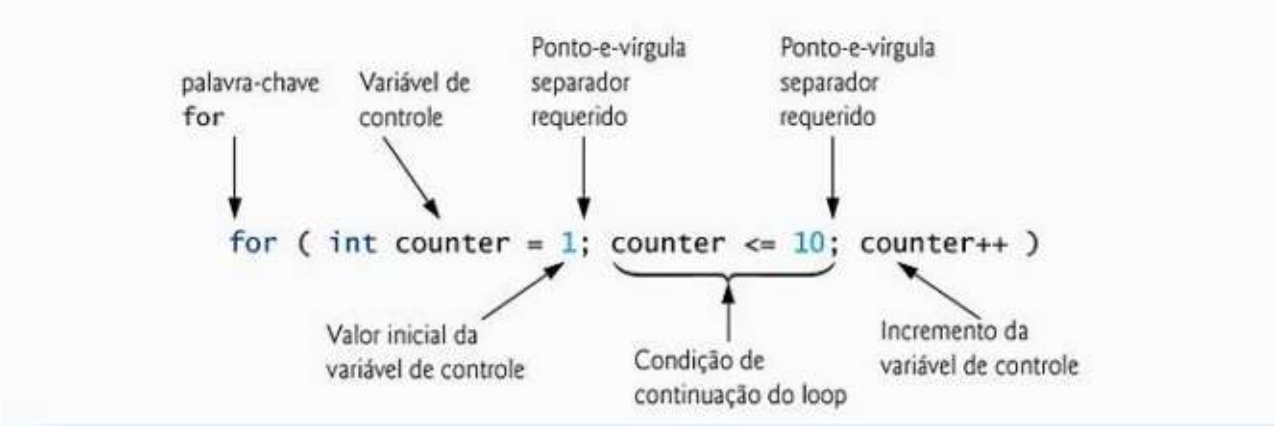
```
int i = 0;
while (i < 10){
    System.out.println(i);
    i++;
}
```

Instrução de repetição for

```
1  // Figura 5.2: ForCounter.java
2  // Repetição controlada por contador com a instrução de repetição for.
3
4  public class ForCounter
5  {
6     public static void main( String[] args )
7     {
8        // cabeçalho da instrução for inclui inicialização,
9        // condição de continuação do loop e incremento
10       for ( int counter = 1; counter <= 10; counter++ )
11          System.out.printf( "%d  ", counter );
12
13       System.out.println(); // imprime uma nova linha
14    } // fim de main
15 } // fim da classe ForCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

Instrução de repetição for

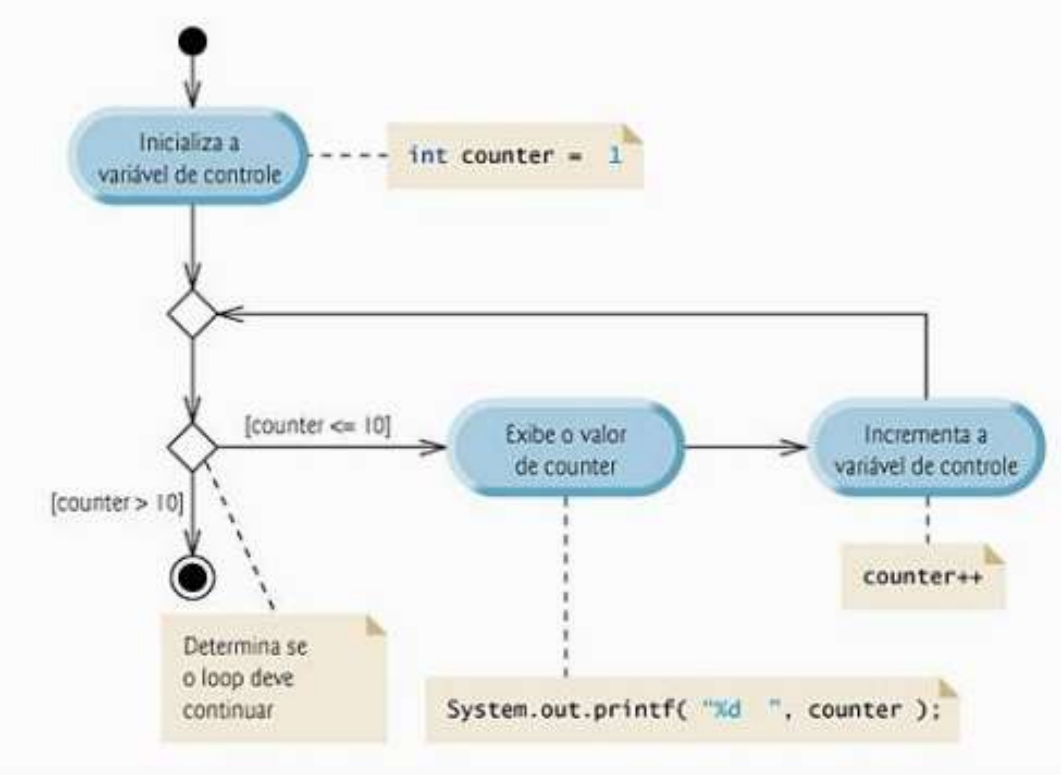



```
for ( inicialização; condiçãoDeContinuaçãoDoLoop; incremento )
   instrução
```

=

```
inicialização;

while ( condiçãoDeContinuaçãoDoLoop )
{
   instrução
   incremento;
}
```

Instrução de repetição for



Instrução de repetição for EXEMPLOS:

a) Varie a variável de controle de 1 a 100 em incrementos de 1.

```
for ( int i = 1; i <= 100; i++ )
```

b) Varie a variável de controle de 100 a 1 em decrementos de 1.

```
for ( int i = 100; i >= 1; i-- )
```

c) Varie a variável de controle de 7 a 77 em incrementos de 7.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Varie a variável de controle de 20 a 2 em decrementos de 2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Varie a variável de controle pelos valores 2, 5, 8, 11, 14, 17, 20.

```
for ( int i = 2; i <= 20; i += 3 )
```

f) Varie a variável de controle pelos valores 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int i = 99; i >= 0; i -= 11 )
```

Instrução de repetição for EXEMPLOS:

Aplicativo: somando os inteiros pares de 2 a 20

Agora consideramos dois aplicativos de exemplo que demonstram as utilizações simples de `for`. O aplicativo na Figura 5.5 utiliza uma instrução `for` para somar os inteiros pares de 2 a 20 e armazenar o resultado em uma variável `int` chamada `total`.

```
1  // Figura 5.5: Sum.java
2  // Somando inteiros com a instrução for.
3
4  public class Sum
5  {
6     public static void main( String[] args )
7     {
8        int total = 0; // inicializa o total
9
10       // total de inteiros pares de 2 a 20
11       for ( int number = 2; number <= 20; number += 2 )
12          total += number;
13
14       System.out.printf( "Sum is %d\n", total ); // exibe os resultados
15    } // fim de main
16 } // fim da classe Sum
```

```
Sum is 110
```

Declarações importantes:

DECLARAÇÃO break

A declaração **break** é utilizada juntamente com a declaração **switch** para fazer o escape desse bloco quando alguma das opções do switch forem executadas; caso contrário o switch continuaria sua execução. Não é muito usual, mas você pode utilizar o **break** com a mesma finalidade (interromper o bloco de comandos e sair da estrutura de controle) para as instruções **for, while ou do-while.**
O trecho de código abaixo exemplifica uma implementação da declaração **break** quando utilizada junto com o switch, anteriormente visto.

DECLARAÇÃO continue

A declaração continue é utilizada para saltar da iteração corrente de uma estrutura de repetição para a próxima iteração. Assim, pode ser utilizada tanto com for quanto **com while ou do-while.**

DECLARAÇÃO return

A declaração **return** é utilizada para sair de um método que está sendo executado ao final

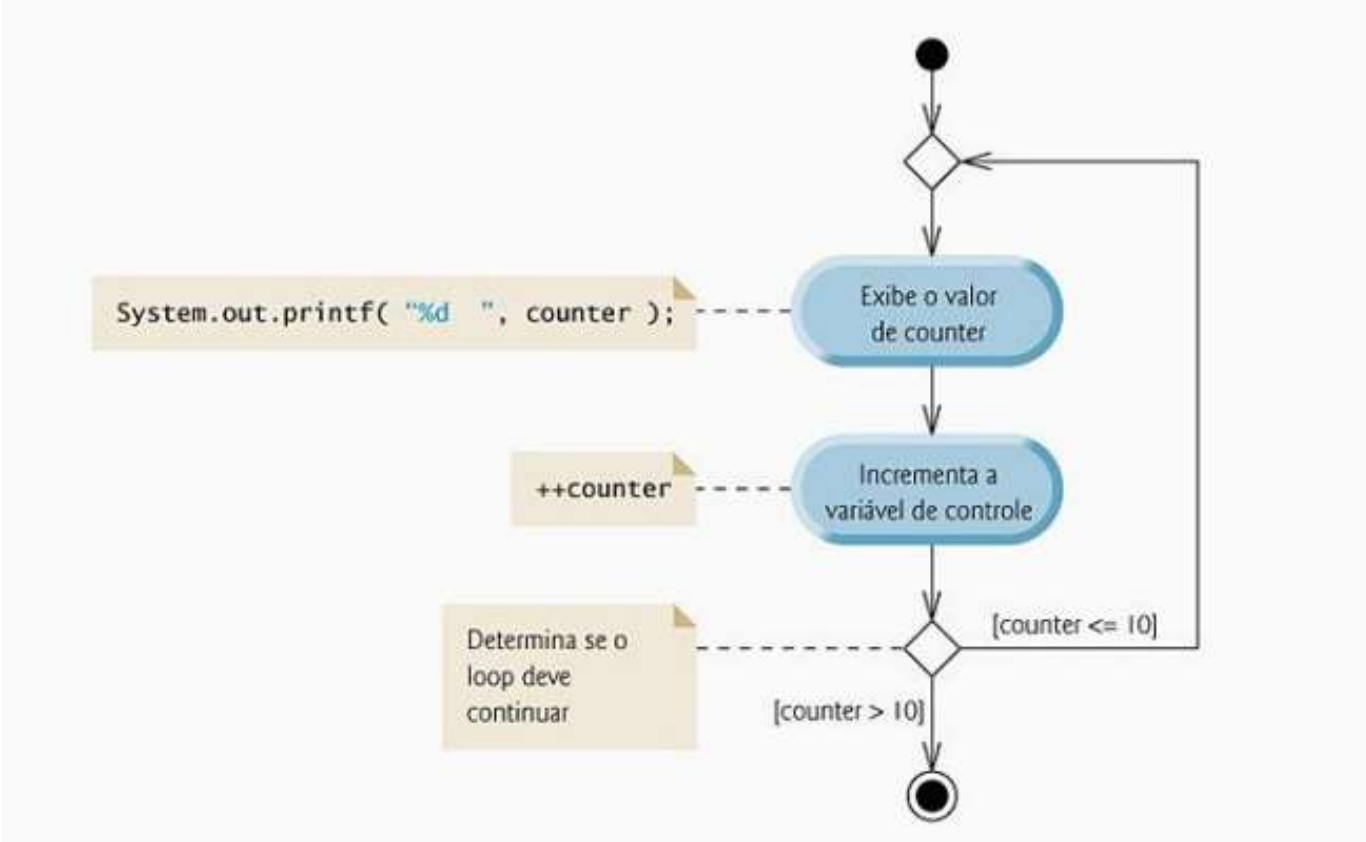
Instrução de repetição do..while:

O **do-while** executa um bloco de comandos repetidamente enquanto a condição lógica imposta como teste for verdadeira (expressão lógica igual a true); a diferença entre ele e o **while** é que o teste da expressão lógica vem depois do bloco; assim, a instrução será executada pelo menos uma vez e somente depois será avaliada.
A declaração **do-while** possui a seguinte sintaxe:

do {
instrução1;
instrução2;
...
} while (expressão_lógica);

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

Instrução de repetição do..while:



Instrução de repetição do..while:

```
public class DoWhileTest
{
   public static void main( String[] args )
   {
      int counter = 1; // inicializa o contador

      do
      {
         System.out.printf( "%d  ", counter );
         ++counter;
      } while ( counter <= 10 ); // fim da instrução do...while

      System.out.println(); // gera saída de um caractere nova linha
   } // fim de main
} // fim da classe DoWhileTest
```

```
1  2  3  4  5  6  7  8  9  10
```

Questão 47 – CAP 2011 – Prova Amarela

47) Analise o código em JAVA a seguir.

```
package prova;
public class Main {

    public static void main(String[] args) {
        int i = 1;
        int a = 0;
        while (i < 10) {
           a++;
           i = i + 1;
           if (a>5) i++;
        }
        System.out.println(i);
        System.out.println(a);
    }
}
```

Em relação ao código acima, assinale a opção correta referente aos valores impressos de i e de a, respectivamente.

(A) 11 e 8
(B) 11 e 7
(C) 10 e 6
(D) 10 e 7
(E) 11 e 7

CAP 2012- Amarela

43) Considerando a construção de algoritmos no que tange às estruturas básicas de controle, analise o programa em JAVA abaixo, desenvolvido no ambiente NetBeans 7.2.1:

```
package p6;
public class P6 { // NetBeans IDE 7.2.1 || D6 1018
    public static void main(String[] args) {
        int c=20;
        while (c <=20 )
        {
            ++c;
        }
        System.out.println(c);
    }
}
```

Dentre as opções abaixo, qual apresenta o valor que será impresso por esse programa ao final de sua execução?

(A) 22
(B) 21
(C) 20
(D) 19
(E) 10

CAP 2011- Amarela

22) Analise o código em JAVA a seguir.

```
package prova;
public class Main {

    public static void main(String[] args) {
        int i = 1;
        int a = 0;
        while (i < 10) {
            ++a;
            i = i + 1;
            if (a>6) ++i;
        }
        System.out.println(i);
        System.out.println(a);
    }
}
```

Sabendo-se que o código acima foi escrito e executado utilizando o IDE NetBeans 6.0.1, assinale a opção correta referente aos valores impressos de i e de a, respectivamente.

(A) 11 e 8
(B) 11 e 7
(C) 10 e 6
(D) 10 e 7
(E) 11 e 7

CLASSES, ATRIBUTOS E MÉTODOS

JAVA – EAGS SIN 2022

Classe	• Representação de um conjunto de objetos com características afins. Definição do comportamento dos objetos (métodos) e seus atributos (atributos).
Objeto	• Uma instância de uma classe. • Armazenamento de estados através de seus atributos e reação a mensagens enviadas por outros objetos.
Herança	• Mecanismo pela qual uma classe (sub-classe) pode estender outra classe (super-classe), estendendo seus comportamentos e atributos.
Polimorfismo	• Princípio pelo qual as instâncias de duas classes ou mais classes derivadas de uma mesma super-classe podem invocar métodos com a mesma assinatura, mas com comportamentos distintos.
Encapsulamento	• Proibição do acesso direto ao estado de um objeto, disponibilizando apenas métodos que alterem esses estados na interface pública.

Na prática

Representação de uma Classe

PESSOA
+ Id:int() + Nome:String(50) + End:String(50)
+CadastrarPessoa() +ListarPessoa() +ExcluirPessoa()

No código:

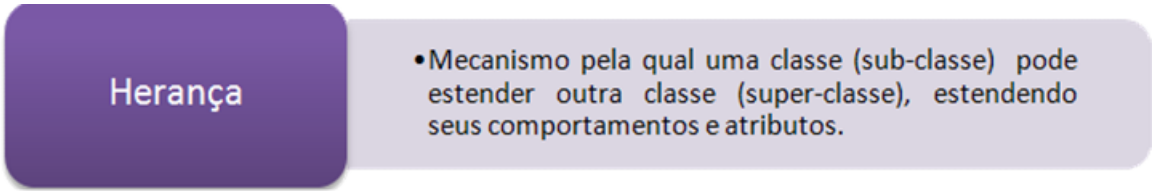
JAVA:

```
public class Pessoa {
 public String Id;
 public String nome;
 public String end;

}
```

OS PILARES DA POO

OS PILARES DA POO:
Herança

HERANÇA



A herança permite À classe que está herdando redefinir qualquer comportamento do que não goste. Tal recurso permite que você adapte seu software, quando seus requisitos mudarem.

"É um" *versus* "tem um": aprendendo quando usar herança

Para apresentar os mecanismos de herança, a primeira seção abordou o que é conhecido como *herança de implementação*. Conforme você viu, a herança de implementação permite que suas classes herdem a implementação de outras classes. Entretanto, somente porque uma classe pode herdar de outra não significa que isso deve ser feito!

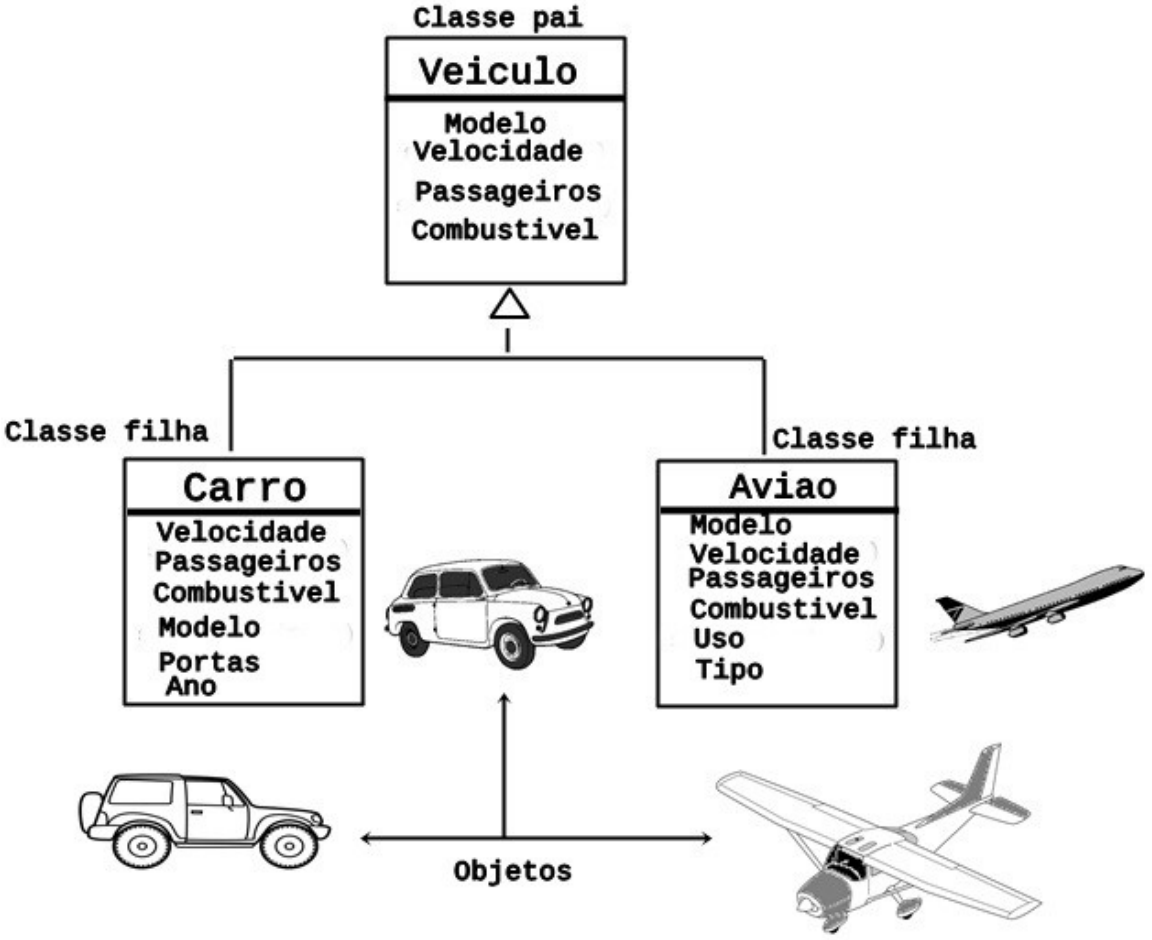
Então, como você sabe quando deve usar herança? Felizmente, existe uma regra geral a ser seguida, para evitar uma herança incorreta.

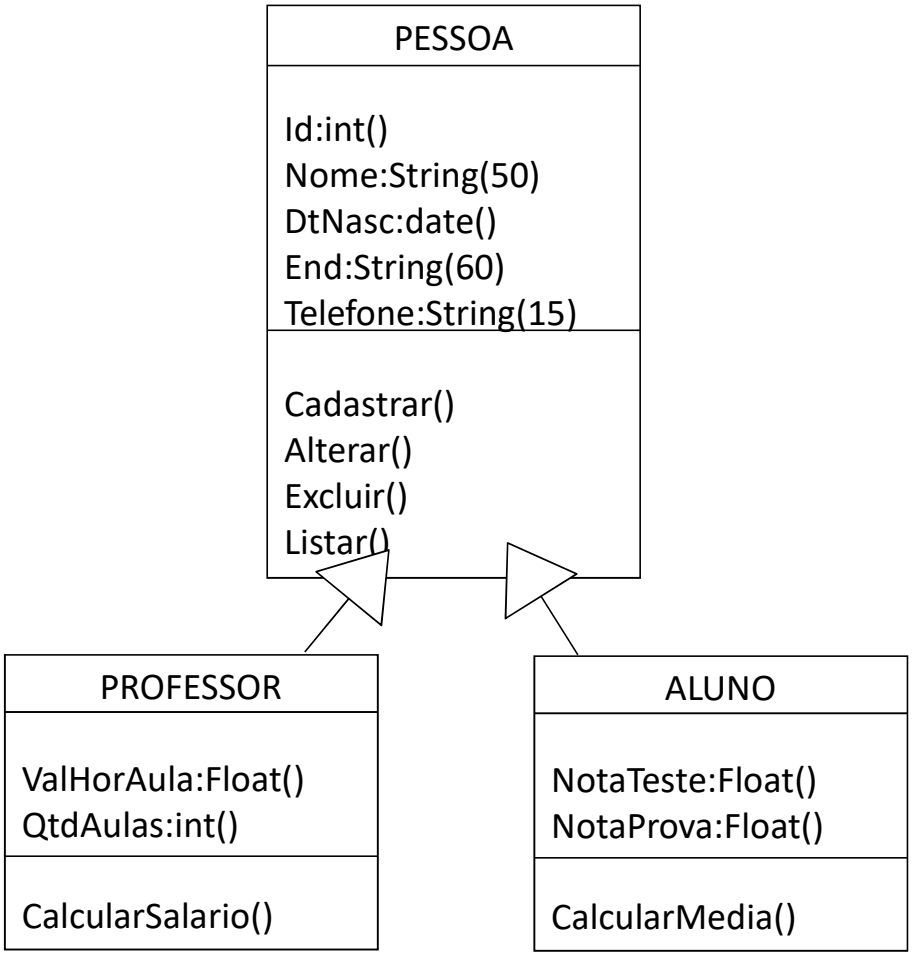
Quando você está considerando a herança para reutilização ou por qualquer outro motivo, precisa primeiro perguntar-se se a classe que está herdando é do mesmo tipo que a classe que está sendo herdada. O fato de pensar em termos de tipo enquanto se herda é freqüentemente referido como teste 'é um'.

NOVO TERMO *É um* descreve o relacionamento em que uma classe é considerada do mesmo tipo de outra.

Para usar 'é um', você diz a si mesmo, "um objeto `CommissionedEmployee` 'é um' `Employee`". Essa declaração é verdadeira e você saberia imediatamente que a herança é válida nessa situação. Agora, pare e considere a interface `Iterator` Java:

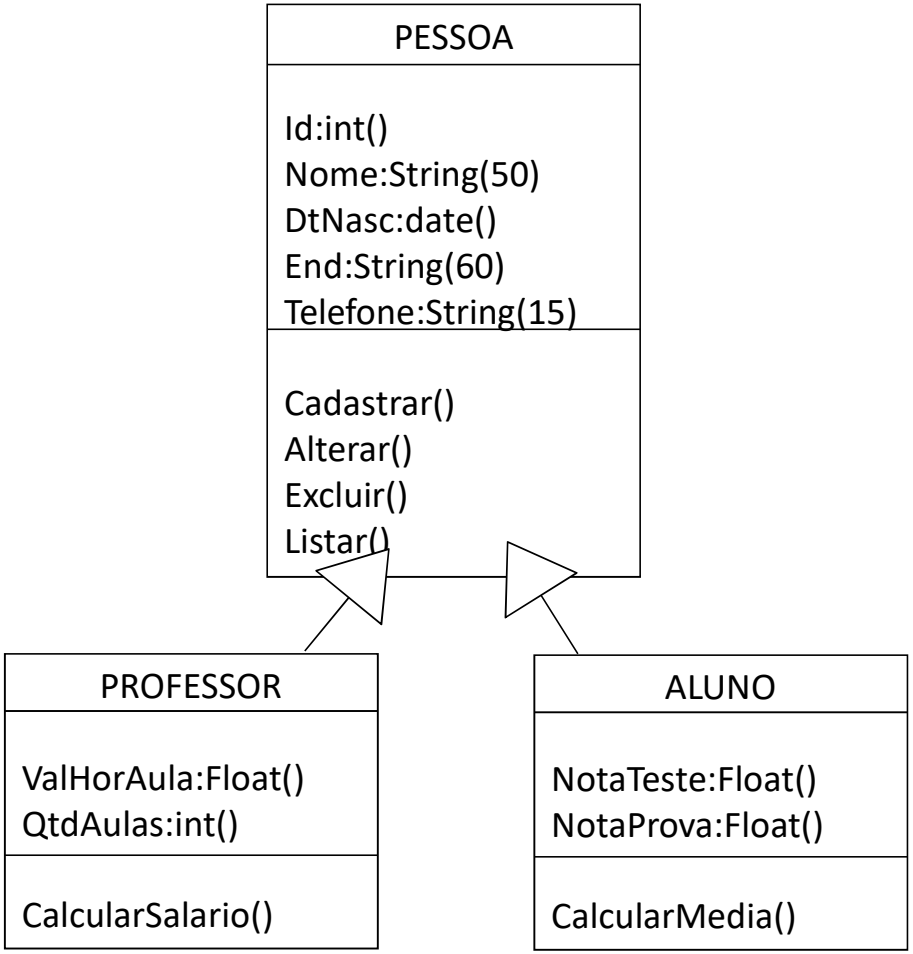
```
public interface Iterator {
      public boolean hasNext();
      public Object next();
      public void remove();
}
```


Classe pai
Veiculo
Modelo
Velocidade
Passageiros
Combustivel
Classe filha
Classe filha
Carro
Velocidade
Passageiros
Combustivel
Modelo
Portas
Ano
Aviao
Modelo
Velocidade
Passageiros
Combustivel
Uso
Tipo
Objetos


PESSOA
Id:int()
Nome:String(50)
DtNasc:date()
End:String(60)
Telefone:String(15)
Cadastrar()
Alterar()
Excluir()
Listar()
PROFESSOR
ValHorAula:Float()
QtdAulas:int()
CalcularSalario()
ALUNO
NotaTeste:Float()
NotaProva:Float()
CalcularMedia()

Na prática

HERANÇA



Código em JAVA:

Pessoa.Java

```java
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

Professor.Java

```java
public class Professor extends Pessoa{
    private double valorHoraAula;
    private int qtdAulas;


    public double getValorHoraAula() {
        return valorHoraAula;
    }

    public void setValorHora(double valorHoraAula) {
        this.valorHoraAula = valorHoraAula;
    }

    public int getQtdAula() {
        return qtdAulas;
    }

    public void setQtdAula(int qtdAulas) {
        this.qtdAulas = qtdAulas;
    }

}
```

OS PILARES DA POO:
Polimorfismo

POLIMORFISMO

Polimorfismo	• Princípio pelo qual as instâncias de duas classes ou mais classes derivadas de uma mesma super-classe podem invocar métodos com a mesma assinatura, mas com comportamentos distintos.

Se o encapsulamento e a herança são os socos um e dois da POO, o polimorfismo é o soco para nocaute seguinte. Sem os dois pilares, você não poderia ter o polimorfismo, e sem o polimorfismo, a POO não seria eficaz. O polimorfismo é onde o paradigma da programação orientada a objetos realmente brilha e seu domínio é absolutamente necessário para a POO eficaz.

Polimorfismo significa muitas formas. Em termos de programação, o polimorfismo permite que um único nome de classe ou nome de método represente um código diferente, selecionado por algum mecanismo automático. Assim, um nome pode assumir muitas formas e como pode representar código diferente o mesmo nome pode representar muitos comportamentos diferentes.

Conceitos sobre Polimorfismo:

1- De inclusão
2- Paramétrico
3- Sobreposição *(Sobrescrita , overwrite)
4- Sobrecarga *(Overhead)

O Polimorfismo de inclusão, às vezes chamado de polimorfismo puro, permite que você trate objetos relacionados genericamente.

Também associado a Sobreposição ou sobrescrita

O Polimorfismo paramétrico permite que você crie métodos e tipos genéricos. Assim como o polimorfismo de inclusão, os métodos e tipos genéricos permitem que você codifique algo uma vez e faça isso trabalhar com muitos tipos diferentes de argumentos.
Também associado a sobrecarga.

Na prática: **SOBRESCRITA**

JAVA:

```java
public class Professor extends Pessoa{
    private double valorHoraAula;
    private int qtdAulas;

    public String getNome() {
        System.out.println("Olá professor: "+nome);
        return nome;
    }

    public double getValorHora() {
        return valorHoraAula;
    }

    public void setValorHora(double valorHoraAula) {
        this.valorHoraAula = valorHoraAula;
    }

    public int getQtdAula() {
        return qtdAulas;
    }

    public void setQtdAula(int qtdAulas) {
        this.qtdAulas = qtdAulas;
    }
```

```java
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {
        System.out.println("Olá aluno: "+nome);
        return nome;
    }
    public double getNota1() {
        return nota1;
    }

    public void setNota1(double nota1) {
        this.nota1 = nota1;
    }

    public double getNota2() {
        return nota2;
    }

    public void setNota2(double nota2) {
        this.nota2 = nota2;
    }
```

Na prática

Pessoa.Java

```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

TesteAppJava.java

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();
        p.setNome("Ana");

        System.out.println("Seja bem vinda" + p.getNome());
    }
}
```

Na prática: **SOBRECARGA**

JAVA:

```
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {...4 linhas }
    public double getNota1() {...3 linhas }

    public void setNota1(double nota1) {...3 linhas }

    public double getNota2() {...3 linhas }

    public void setNota2(double nota2) {...3 linhas }


   public void imprimir(){
        System.out.println("Nome: "+ nome);
        System.out.println("Endereço: "+ end);
        System.out.println("Idade: "+ idade);
        System.out.println("Nota1: "+ nota1);
        System.out.println("Nota2: "+ nota2);

   }

   public void imprimir(double nota1, double nota2){
        double md= (nota1+nota2)/2;
        System.out.println("A média é:"+ md);
   }
```

Na prática: **SOBRECARGA**

JAVA:

Sem nenhum parâmetro!!!

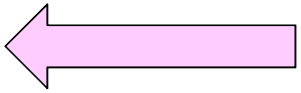
```
10
11      public static void main(String[] args) {
12
13
14          Aluno a = new Aluno();
15
16
17          a.setNome("Matheus");
18          a.setEnd("Rua suvaco da minhoca");
19          a.setIdade(19);
20          a.setNota1(7.9);
21          a.setNota2(10.0);
22
23          a.imprimir();
24
25
26      }
27
```

```
Saída -
Nome: Matheus
Endereço: Rua suvaco da minhoca
Idade: 19
Nota1: 7.9
Nota2: 10.0
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Na prática: **SOBRECARGA**

JAVA:

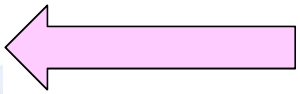
```
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {...4 linhas }
    public double getNota1() {...3 linhas }

    public void setNota1(double nota1) {...3 linhas }

    public double getNota2() {...3 linhas }

    public void setNota2(double nota2) {...3 linhas }


   public void imprimir(){
        System.out.println("Nome: "+ nome);
        System.out.println("Endereço: "+ end);
        System.out.println("Idade: "+ idade);
        System.out.println("Nota1: "+ nota1);
        System.out.println("Nota2: "+ nota2);

   }

   public void imprimir(double nota1, double nota2){
        double md= (nota1+nota2)/2;
        System.out.println("A média é:"+ md);
   }
```



Na prática: **SOBRECARGA**

JAVA:

```
11      public static void main(String[] args) {
12
13
14          Aluno a = new Aluno();
15
16
17          a.setNome("Matheus");
18          a.setEnd("Rua suvaco da minhoca");
19          a.setIdade(19);
20          a.setNota1(7.9);
21          a.setNota2(10.0);
22
23          a.imprimir(a.getNota1(), a.getNota2());
24
25
26      }
27
```



Com 2 parâmetros!!!

```
Saída -
run:
A média é:8.95
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

OS PILARES DA POO:
Encapsulamento

Encapsulamento: o primeiro pilar

Em vez de ver um programa como uma única entidade grande e monolítica, o encapsulamento permite que você o divida em várias partes menores e independentes. Cada parte possui implementação e realiza seu trabalho independentemente das outras partes. O encapsulamento mantém essa independência, ocultando os detalhes internos ou seja, a implementação de cada parte, através de uma interface externa.

NOVO TERMO Encapsulamento é a característica da OO de ocultar partes independentes da implementação. O encapsulamento permite que você construa partes ocultas da implementação do software, que atinjam uma funcionalidade e ocultam os detalhes de implementação do mundo exterior.

Encapsulamento

- Proibição do acesso direto ao estado de um objeto, disponibilizando apenas métodos que alterem esses estados na interface pública.

Público, privado e protegido

O que aparece e o que não aparece na interface pública é governado por diversas palavras chave. Cada linguagem OO define o seu próprio conjunto de palavras-chave, mas fundamentalmente essas palavras-chave acabam tendo efeitos semelhantes:

A maioria das linguagens OO suporta três níveis de acesso:

- Público — Garante o acesso a todos os objetos.
- Protegido — Garante o acesso à instância, ou seja, para aquele objeto, e para todas as subclasses (mais informações sobre subclasses no Dia 4, "Herança: obtendo algo para nada").
- Privado — Garante o acesso apenas para a instância, ou seja, para aquele objeto.

O nível de acesso que você escolhe é muito importante para seu projeto. Todo comportamento que você queira tornar visível para o mundo, precisa ter acesso público. Tudo que você quiser ocultar do mundo exterior precisa ter acesso protegido ou privado.

Na prática

ENCAPSULAMENTO

No código: O uso do private Determina o conceito de Encapsulamento

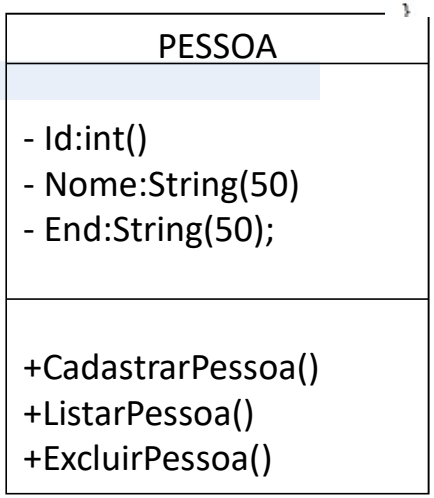
JAVA:

Pessoa.Java

```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;

}
```

TesteAppJava.java

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();

    }
}
```



Na prática

Pessoa.Java

```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

TesteAppJava.java

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();
        p.setNome("Ana");

        System.out.println("Seja bem vinda" + p.getNome());
    }
}
```

PESSOA
- Id:int() - Nome:String(50) -End: String(50)
+getId() +setId() +getNome() ...

Recursividade

O que é recursividade

- Tipicamente em Java os programas são estruturados como objetos que enviam mensagens uns aos outros através de seus métodos de uma forma disciplinada, hierárquica.

```
public static void main(String[] args) {
    A a = new A();
    C c = new C()
    B b = new B(c);

    a.metodoA(b);
}

...
public void metodoA(B b) {
    …
    b.metodoB(getC());
    …
}

...
public void metodoB(C c) {
    …
    c.metodoC();
    ...
}
```

metodoA → metodoB → metodoC
metodoC → metodoB → metodoA (pilha de execução)

Quando um método de um objeto faz uma chamada a ele mesmo, dizemos que este método é recursivo.

Um método recursivo é, na verdade, capaz de resolver o caso mais simples do problema que se busca solucionar, ou seja, o caso base. Em outras palavras, quando o método é chamado para o caso base, ele retorna um resultado. Quando ele é chamado para resolver um caso mais complexo ele divide o problema em duas partes:

→ Um pedaço que ele sabe resolver (caso base);

→ Um pedaço que ele não sabe resolver por completo, mas sabe resolver uma parte, que na verdade lembra o problema original, só que em uma versão mais simples. Assim, neste momento o método chama ele mesmo para resolver a versão mais simples. Cada vez que o método chama ele mesmo, dizemos que ele deu um passo de recursão

A sequencia de chamadas ao método para problemas cada vez menores dele mesmo irá convergir para o caso base, para o qual o método tem uma solução e retornará um resultado, não sendo mais necessários passos de recursão.

Exemplo: o que é um diretório em um computador? Um diretório pode estar vazio, conter arquivos, ou conter outros diretórios (que por sua vez, enquanto diretório pode estar vazio, conter arquivos ou outros diretórios, que por sua vez...).

Como faríamos para listar todos os arquivos de um diretório, incluindo os arquivos dos seus subdiretórios?
→ Um método que recebe um diretório e lê os arquivos deste diretório
→ Chamadas recursivas a este método precisariam ser realizadas sempre que um subdiretório for encontrado.
→ O caminhamento em árvores é tipicamente uma atividade recursiva.

```
listaArquivos(Arquivo D) {
    for( Arquivo a : D ) {
        if( a.isDiretorio() ) {
            return listaArquivos(a); //chamada recursiva, para resolver um problema menor
        }
        else {
            System.out.println(a.getNomeCompleto()); //caso base
        }
}
```

Outro exemplo: fatoriais

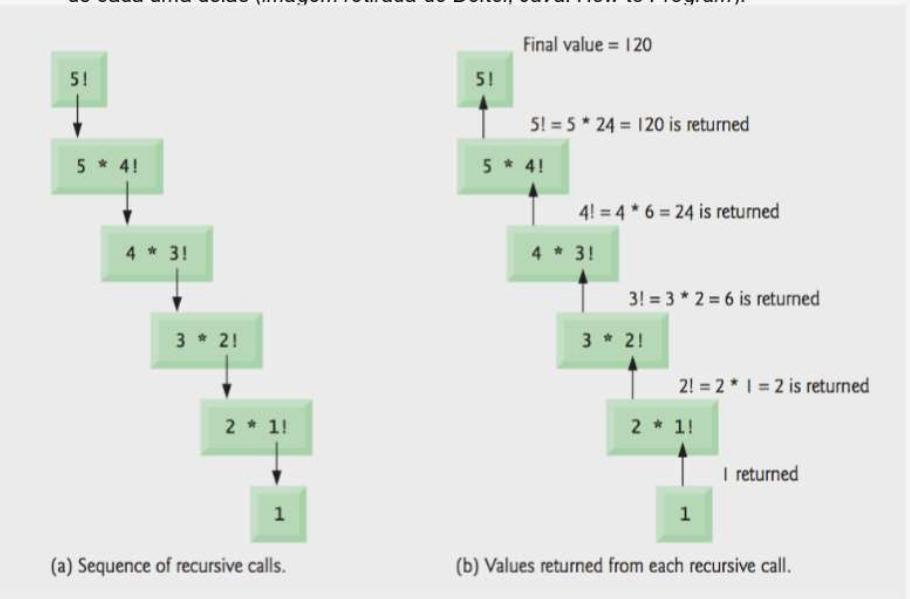
- Relembrando: n! = n . (n-1) . (n-2) . … . 1; 1! = 0! =1
- 5! = 5 . 4 . 3 . 2 . 1 = 120
- Podemos calcular isso de forma iterativa:

```
public void fatorial( int numero ) {
    int fatorial = 1;
    for ( int i = numero; i >= 1; i-- )
        fatorial *= i;
}
```

- Mas também podemos pensar no fatorial como um modelo recursivo: n! = n . (n-1)!
  - 5! = 5 . 4!
- Neste caso, teríamos uma solução diferente:

```
public void fatorial( int numero ) {
   if( numero <= 1 )  //testa se é caso base
      return 1;
   else  //passo de recursão
      return numero * fatorial( numero - 1 );
}
```

- Se chamamos fatorial(5) veja abaixo a sequencia de chamadas recursivas e o retorno de cada uma delas (*imagem retirada de Deitel, Java: How to Program*):



```
public void fatorial( int numero ) {
        if( numero <= 1 ) //testa se é caso base
        return 1;
                else //passo de recursão
                        return numero * fatorial( numero - 1 );
}
```

→ Se chamamos fatorial(5) veja abaixo a sequencia de chamadas recursivas e o retorno de cada uma delas.

→ Erro de recursão infinita: o programador escreve o passo de recursão incorreto, de forma que as chamadas recursivas nunca convergem para o caso base.

1.3 Recursividade

Um tipo especial de procedimento será utilizado, algumas vezes, ao longo do texto. É aquele que contém, em sua descrição, uma ou mais chamadas a si mesmo. Um procedimento dessa natureza é denominado *recursivo*, e a chamada a si mesmo é dita *chamada recursiva*. Naturalmente, todo procedimento, recursivo ou não, deve possuir pelo menos uma chamada proveniente de um local exterior a ele. Essa chamada é denominada *externa*. Um procedimento não recursivo é, pois, aquele em que todas as chamadas são externas.

De modo geral, a todo procedimento recursivo corresponde um outro não recursivo que executa, exatamente, a mesma computação. Contudo, a recursividade pode apresentar vantagens concretas. Freqüentemente, os procedimentos recursivos são mais concisos do que um não recursivo correspondente. Além disso, muitas vezes é aparente a relação direta entre um procedimento recursivo e uma prova por indução matemática. Nesses casos, a verificação da correção pode se tornar mais simples. Entretanto, muitas vezes há desvantagens no emprego prático da recursividade. Um algoritmo não recursivo equivalente pode ser mais eficaz.

O exemplo clássico mais simples de recursividade é o cálculo do fatorial de um inteiro $n \geq 0$. Um algoritmo recursivo para efetuar esse cálculo encontra-se descrito em seguida. A idéia do algoritmo é muito simples. Basta observar que o fatorial de n é n vezes o fatorial de $n-1$, para $n > 0$. Por convenção, sabe-se que $0! = 1$. No algoritmo a seguir, as chamadas recursivas são representadas pela função fat. A chamada externa é $fat(n)$.

algoritmo 1.2: fatorial (recursivo)
função $fatorial(i)$
$fat(i) :=$ **se** $i \leq 1$ **então** 1 **senão** $i \times fat(i-1)$

Para efeito de comparação, o algoritmo 1.3 descreve o cálculo do fatorial de n de forma não recursiva. A variável fat representa, agora, um vetor e não mais uma função. O elemento $fat[n]$ contém, no final, o valor do fatorial desejado.

algoritmo 1.3: fatorial (não recursivo)
$fat[0] := 1$
para $j := 1, \ldots, n$ **faça**
$fat[j] := j \times fat[j-1]$

Um exemplo conhecido, onde a solução recursiva é natural e intuitiva, é o do *Problema da Torre de Hanói*. Este consiste em três pinos, A, B e C, denominados *origem*, *destino* e *trabalho*, respectivamente, e n discos de diâmetros diferentes. Inicialmente, todos os discos se encontram empilhados no pino-origem, em ordem decrescente de tamanho, de baixo para cima. O objetivo é empilhar todos os discos no pino-destino, atendendo às seguintes restrições: (i) apenas um disco pode ser movido de cada vez, e (ii) qualquer disco não pode ser jamais colocado sobre outro de tamanho menor.

A solução do problema é descrita a seguir. Naturalmente, para $n > 1$, o pino-trabalho deverá ser utilizado como área de armazenamento temporário. O raciocínio utilizado para resolver o problema é semelhante ao de uma prova matemática por indução. Suponha que se saiba como resolver o problema até $n-1$ discos, $n > 1$, de forma recursiva. A extensão para n discos pode ser obtida pela realização dos seguintes passos.

Sequência de Fibonacci em Java

A Sequência de Fibonacci tem como primeiros termos os números 0 e 1 e, a seguir, cada termo subsequente é obtido pela soma dos dois termos predecessores:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Nenhuma outra sequência de números foi tão estudada e possui aplicações em áreas tão distintas, como por exemplo Biologia, Arquitetura, Arte e outras. A razão entre dois números consecutivos da sequência converge para um valor constante de 1,618... conhecido como número de ouro, número áureo ou proporção áurea (golden ratio). Este número é muito "apreciado" por artistas e cientistas porque está presente em diversos fenômenos da natureza

Os números da sequência de Fibonacci podem ser gerados por uma regra (recorrência) simples:

Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

Recursividade

IMPLEMENTAÇÃO 1 - RECURSIVA TRADICIONAL
Consiste na tradução direta da recorrência para um algoritmo recursivo.

```
public class Fibonacci {

    static long fibo(int n) {
        if (n < 2) {
            return n;
        } else {
            return fibo(n - 1) + fibo(n - 2);
        }
    }

    public static void main(String[] args) {

        // teste do programa. Imprime os 30 primeiros termos
        for (int i = 0; i < 30; i++) {
            System.out.print("(" + i + "):" + Fibonacci.fibo(i) + "\t");
        }

    }

}
```

Sequência de Fibonacci em Java

IMPLEMENTAÇÃO 2 - RECURSIVA UTILIZANDO O OPERADOR TERNÁRIO
Com o uso do operador ternário do Java é possível construir uma implementação bem mais compacta do método recursivo.

```
public class Fibonacci {

    static long fibo(int n) {
        return (n < 2) ? n : fibo(n - 1) + fibo(n - 2);
    }


    public static void main(String[] args) {

         // teste do programa. Imprime os 30 primeiros termos
        for (int i = 0; i < 30; i++) {
            System.out.print("(" + i + "):" + Fibonacci.fibo(i) + "\t");
        }

    }

}
```

Recursividade

18.4 Exemplo que utiliza recursão: série de Fibonacci

A **série de Fibonacci**

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inicia com 0 e 1 e tem a propriedade de que cada número de Fibonacci subsequente é a soma dos dois anteriores. Essa série ocorre na natureza e descreve a forma de uma espiral. A relação de números de Fibonacci sucessivos converge para um valor constante de 1,618..., um número que é chamado de **relação áurea** ou **média áurea.** Humanos tendem a achar a média áurea esteticamente agradável. Os arquitetos frequentemente projetam janelas, salas e edifícios cujo comprimento e largura estão na relação da média áurea. Cartões postais são frequentemente projetados com uma relação de comprimento/largura igual à relação áurea.

A série de Fibonacci pode ser definida recursivamente como segue:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

Observe que há *dois casos básicos* para o cálculo de Fibonacci: `fibonacci(0)` é definido como 0, e `fibonacci(1)` como 1. O programa na Figura 18.5 calcula recursivamente o i-ésimo número de Fibonacci, utilizando o método `fibonacci` (linhas 10–18). O método `main` (linhas 21–26) testa `fibonacci`, exibindo os valores de Fibonacci de 0–40. A variável `counter` criada no cabeçalho `for` (linha 23) indica qual número de Fibonacci calcular para cada iteração do loop. Os números Fibonacci tendem a se tornar rapidamente grandes (embora não tão rapidamente como os fatoriais). Portanto, utilizamos o tipo `BigInteger` como o tipo de parâmetro e o tipo de retorno do método `fibonacci`.

```
1  // Figura 18.5: FibonacciCalculator.java
2  // Método recursivo de Fibonacci.
3  import java.math.BigInteger;
4
5  public class FibonacciCalculator
6  {
7     private static BigInteger TWO = BigInteger.valueOf( 2 );
8
```

Recursividade

```
1  // Figura 18.5: FibonacciCalculator.java
2  // Método recursivo de Fibonacci.
3  import java.math.BigInteger;
4
5  public class FibonacciCalculator
6  {
7     private static BigInteger TWO = BigInteger.valueOf( 2 );
8
9     // declaração recursiva do método fibonacci
10    public static BigInteger fibonacci( BigInteger number )
11    {
12       if ( number.equals( BigInteger.ZERO ) ||
13            number.equals( BigInteger.ONE ) ) // casos básicos
14          return number;
15       else // passo de recursão
16          return fibonacci( number.subtract( BigInteger.ONE ) ).add(
17             fibonacci( number.subtract( TWO ) ) );
18    } // fim do método fibonacci
19
20    // exibe os valores de fibonacci de 0-40
21    public static void main( String[] args )
22    {
23       for ( int counter = 0; counter <= 40; counter++ )
24          System.out.printf( "Fibonacci of %d is: %d\n", counter,
25             fibonacci( BigInteger.valueOf( counter ) ) );
26    } // fim de main
27 } // fim da classe FibonacciCalculator
```

Recursividade

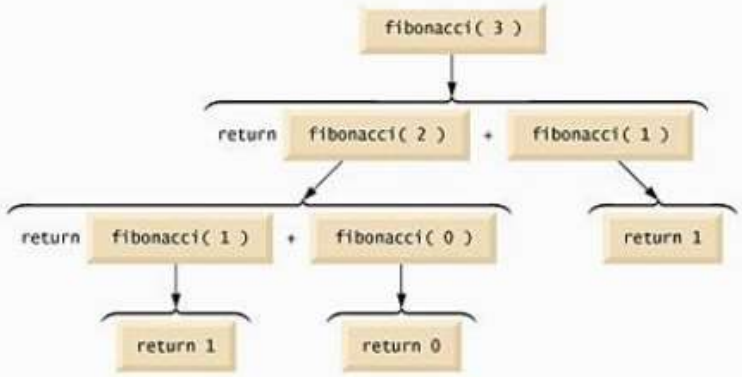
```
Fibonacci of 0 is: 0
Fibonacci of 1 is: 1
Fibonacci of 2 is: 1
Fibonacci of 3 is: 2
Fibonacci of 4 is: 3
Fibonacci of 5 is: 5
Fibonacci of 6 is: 8
Fibonacci of 7 is: 13
Fibonacci of 8 is: 21
Fibonacci of 9 is: 34
Fibonacci of 10 is: 55
...
Fibonacci of 37 is: 24157817
Fibonacci of 38 is: 39088169
Fibonacci of 39 is: 63245986
Fibonacci of 40 is: 102334155
```

Figura 18.5 | Os números de Fibonacci gerados com um método recursivo.

Recursividade

Analisando as chamadas para o método Fibonacci

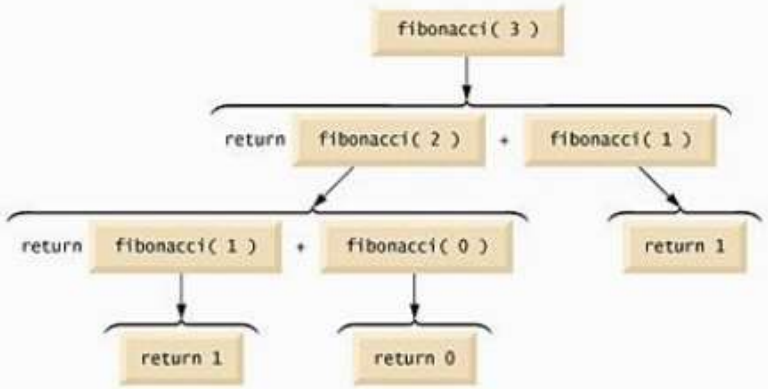
A Figura 18.6 mostra como o método `fibonacci` avalia `fibonacci(3)`. Observe que, na parte inferior da figura, resta-nos os valores 1, 0 e 1 — os resultados da avaliação dos casos básicos. Os primeiros dois valores de retorno (da esquerda para direita), 1 e 0, são retornados como os valores das chamadas `fibonacci(1)` e `fibonacci(0)`. A soma 1 mais 0 é retornada como o valor de `fibonacci(2)`. Isso é adicionado ao resultado (1) da chamada para `fibonacci(1)`, produzindo o valor 2. Esse valor final é então retornado como o valor de `fibonacci(3)`.



Recursividade

Analisando as chamadas para o método Fibonacci

A Figura 18.6 mostra como o método `fibonacci` avalia `fibonacci(3)`. Observe que, na parte inferior da figura, resta-nos os valores 1, 0 e 1 — os resultados da avaliação dos casos básicos. Os primeiros dois valores de retorno (da esquerda para direita), 1 e 0, são retornados como os valores das chamadas `fibonacci(1)` e `fibonacci(0)`. A soma 1 mais 0 é retornada como o valor de `fibonacci(2)`. Isso é adicionado ao resultado (1) da chamada para `fibonacci(1)`, produzindo o valor 2. Esse valor final é então retornado como o valor de `fibonacci(3)`.



Recursividade – Fatorial

18.3 Exemplo que utiliza recursão: fatoriais

Vamos escrever um programa recursivo para efetuar um cálculo matemático popular. Considere o fatorial de um inteiro positivo n, escrito $n!$ (pronuncia-se "n fatorial"), que é o produto

$$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

com 1! igual a 1 e 0! definido como 1. Por exemplo, 5! é o produto $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que é igual a 120.

O fatorial de inteiro `number` (onde `number` ≥ 0) pode ser calculado iterativamente (não recursivamente) utilizando-se uma instrução `for` como a seguinte:

```
factorial = 1;

for ( int counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

Chega-se a uma declaração recursiva do método fatorial observando o seguinte relacionamento:

$$n! = n \cdot (n - 1)!$$

Por exemplo, 5! é claramente igual a $5 \cdot 4!$, como mostrado pelas seguintes equações:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

A avaliação de 5! prosseguiria como mostrado na Figura 18.2. A Figura 18.2(a) mostra como a sucessão de chamadas recursivas prossegue até 1! (o caso básico) é avaliado como 1, que termina a recursão. A Figura 18.2(b) mostra os valores retornados de cada chamada recursiva para seu chamador até que o valor final seja calculado e retornado.

Recursividade – Fatorial

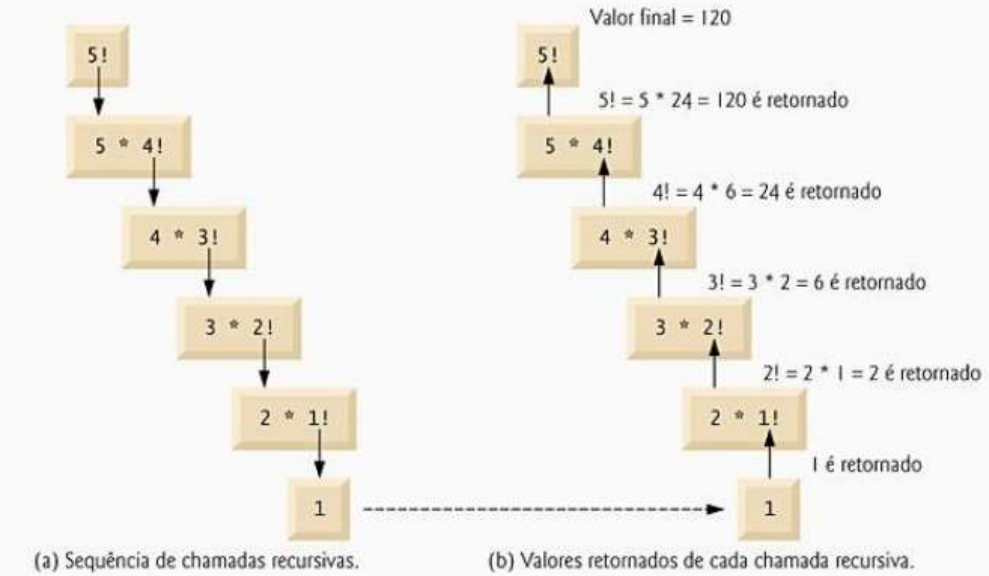


Figura 18.2 | Avaliação recursiva de 5!

Recursividade – Fatorial

```
1  // Figura 18.3: FactorialCalculator.java
2  // Método fatorial recursivo.
3
4  public class FactorialCalculator
5  {
6     // método recursivo fatorial (assume que seu parâmetro é >= 0)
7     public long factorial( long number )
8     {
9        if ( number <= 1 ) // testa caso básico
10          return 1; // casos básicos: 0! = 1 e 1! = 1
11       else // passo de recursão
12          return number * factorial( number - 1 );
13    } // fim do método factorial
14
15    // gera saída de fatoriais para valores 0-21
16    public static void main( String[] args )
17    {
18       // calcula o fatorial de 0 a 21
19       for ( int counter = 0; counter <= 21; counter++ )
20          System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
21    } // fim de main
22 } // fim da classe FactorialCalculator
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
...
12! = 479001600 —— 12! causes overflow for int variables
...
20! = 2432902008176640000
```

2020

62 – Considere as representações UML das classes abaixo e assinale a alternativa que possui dois atributos privados.

a)

MilitarAeronautica
idade : int + posto : String # especialidade : String

b)

MilitarAeronautica
- idade : int - posto : String + especialidade : String

c)

MilitarAeronautica
+ idade : int - posto : String # especialidade : String

d)

MilitarAeronautica
+ idade : int + posto : String # especialidade : String

2020

62 – Considere as representações UML das classes abaixo e assinale a alternativa que possui dois atributos privados.

a)

MilitarAeronautica
idade : int + posto : String # especialidade : String

b)

MilitarAeronautica
- idade : int - posto : String + especialidade : String

c)

MilitarAeronautica
+ idade : int - posto : String # especialidade : String

d)

MilitarAeronautica
+ idade : int + posto : String # especialidade : String

B

2020

82 – Considere as representações UML das classes abaixo e assinale a alternativa que apresenta uma sobrecarga de métodos.

a)

ComparadorNumerico
+ max(x : int, y : int) : int + max(x : long, y : long) : long

b)

ComparadorNumerico
+ max(x : int, y : int) : int + min(x : int, y : int) : int

c)

ComparadorNumerico
+ max(x : int, y : int) : int + min(x : int) : void

d)

ComparadorNumerico
+ max(x : int, y : int) : int + maior(x : int, y : int) : int

2020

82 – Considere as representações UML das classes abaixo e assinale a alternativa que apresenta uma sobrecarga de métodos.

a)

ComparadorNumerico
+ max(x : int, y : int) : int + max(x : long, y : long) : long

b)

ComparadorNumerico
+ max(x : int, y : int) : int + min(x : int, y : int) : int

c)

ComparadorNumerico
+ max(x : int, y : int) : int + min(x : int) : void

d)

ComparadorNumerico
+ max(x : int, y : int) : int + maior(x : int, y : int) : int

A

JAVA – EAGS SIN 2022

2022

48 – Em relação à Linguagem de Programação JAVA, coloque V para verdadeiro e F para falso. Em seguida, assinale a alternativa com a sequência correta.

( ) A linguagem JAVA distingue letras maiúsculas de letras minúsculas na criação da variável.
( ) A linguagem JAVA possui algumas palavras reservadas que não podem ser utilizadas, por exemplo: super e import.
( ) As variáveis precisam ter o símbolo $ (cifrão) na frente.
( ) Nas classes internas, assim como nas classes normais, para acessar os atributos da classe interna, utiliza-se a palavra-chave **this**.

a) V - V - F - V
b) F - V - V - F
c) F - F - F - V
d) V - V - F - F

A