Introdução Java

História do JAVA

> Em 1991, um grupo de engenheiros da **Sun Microsystems** foi encarregado de criar uma nova linguagem que pudesse ser utilizada em pequenos equipamentos como controles de TV, telefones, fornos, geladeiras, etc. Essa linguagem deveria dar a esses aparelhos a capacidade de se comunicar entre si, para que a casa se comportasse como uma federação. Deveria ainda ser capaz de gerar códigos muito pequenos, que pudessem ser executados em vários aparelhos diferentes, e praticamente infalível.
> Os engenheiros escolheram o **C++** como ponto de partida. Orientada a objetos, muito poderosa e gerando pequenos programas, parecia a escolha correta. Para solucionar o problema da execução em várias arquiteturas, eles utilizaram o conceito da máquina virtual, onde cada fabricante iria suportar algumas funções básicas que os programas utilizariam.
> Até hoje a linguagem resultante deste projeto não é utilizada em aparelhos eletrodomésticos. Ao invés disso, o Java se tornou um das linguagens de programação mais utilizadas no planeta.

Plataformas JAVA

JAVAava SE: plataforma básica destinada ao desenvolvimento da maior parte das aplicações desktop que rodam nas estações de trabalho.
A Oracle distribui a Java SE na forma de um JDK (Java Development Kit), vem com ferramentas para compilação, execução, debugging, geração de documentação (javadoc), empacotador de componentes, bibliotecas comuns e etc.

JAVA EE: A Java EE (Java Platform, Enterprise Edition) é uma plataforma padrão para desenvolver aplicações Java de grande porte e/ou para a internet, que inclui bibliotecas e funcionalidades para implementar software Java distribuído, baseado em componentes modulares que executam em servidores de aplicações e que suportam escalabilidade, segurança, integridade e outros requisitos de aplicações corporativas ou de grande porte.

Java ME: A Java ME (Java Platform, Micro Edition) é uma plataforma que possibilita a criação de sistemas embarcados em dispositivos compactos, como celulares, PDAs, controles remotos, etc.

Java Card, JavaFX e Java TV: A tecnologia **Java Card** permite que aplicações Java sejam executadas com segurança
em cartões inteligentes ou outros dispositivos similares.
A plataforma **JavaFX** é usada para desenvolver aplicações ricas (no sentido de bonitas e interativas) para a internet, que podem ser executadas em uma ampla variedade de dispositivos conectados, como em ambientes desktop, internet (browser), telefones celulares e TVs.
A **Java TV** fornece APIs para desenvolvimento de aplicações que são executadas em set-up boxes (receptores de TV digital), players de Blu-ray e outros dispositivos de mídia digital.

JVM (Java Virtual Machine)

O Java utiliza do conceito de **máquina virtual**, se localiza entre o sistema operacional e a aplicação, uma camada extra que traduz todos os processos da sua aplicação com as respectivas chamadas do sistema operacional, onde ele esta rodando.

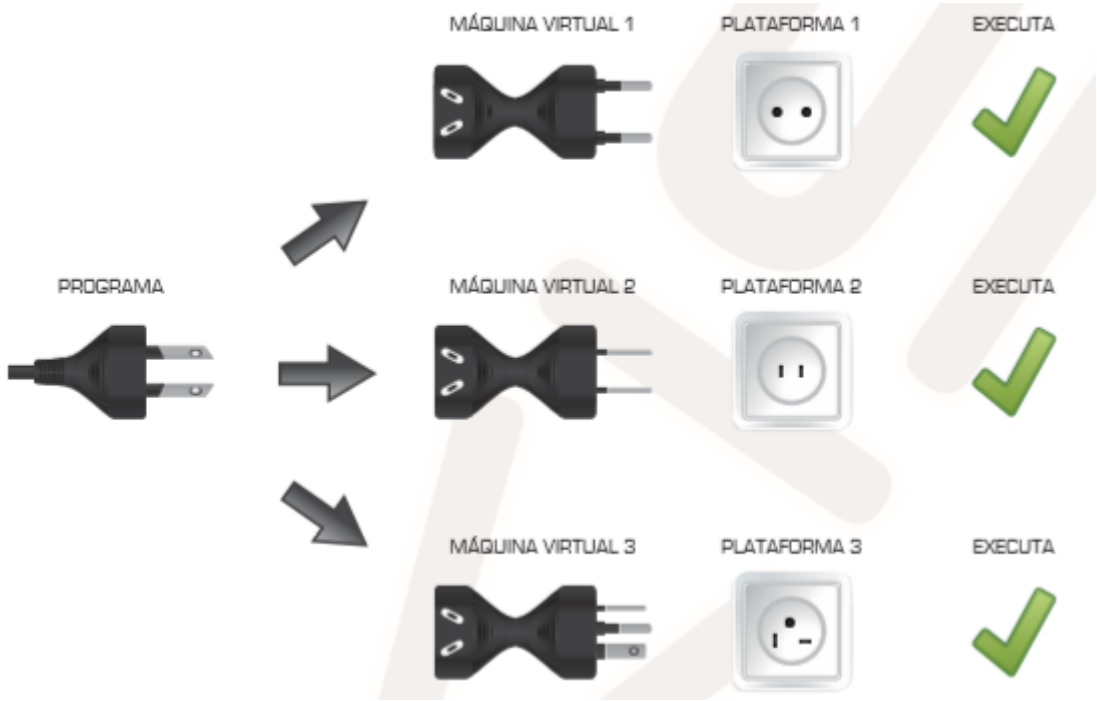


Figura 1: Funcionamento da máquina virtual

Ambiente de Trabalho JAVA

Para começar a desenvolver na programação Java, primeiramente você tem que instalar o **JDK (Java Development Kit)** da plataforma Java SE, que nada mais é do que um kit de desenvolvimento Java com aplicativos para compilar e debugar seus código-fonte.
Para baixar o JDK da Oracle basta ir no site do oracle https://www.oracle.com/java/technologies/javase-jdk8-downloads.html e faça o download da última versão do JDK para o seu sistema operacional.

Figura 2. Download JDK

Instalando a IDE Eclipse

A instalação do eclipse é muito simples. Primeiramente, precisamos ter o arquivo de instalação da IDE, isso é bem fácil de encontrar. Eu estou indicando o site oficial do eclipse que segue o link:
http://www.eclipse.org/downloads/
Quando entrar na página oficial deverá escolher para qual sistema operacional deseja instalar.
Depois que fizer o download você irá executar o instalador que começara a instalação do Eclipse.

Figura 3. Instalador Eclipse

Você irá marcar a primeira opção que é o nosso objetivo programar em Java, as outras opções servem para outras linguagens.

O próximo passo é escolher modificar ou não a página de destino que irá instalar e mesmo criar alguns atalhos para acessar o mesmo. Feito isso clique em *install*:

Figura 4. Criar local de instalação do Eclipse

Na sequência irá aparecer os termos de uso que você terá que apertar em *"Accept Now"*

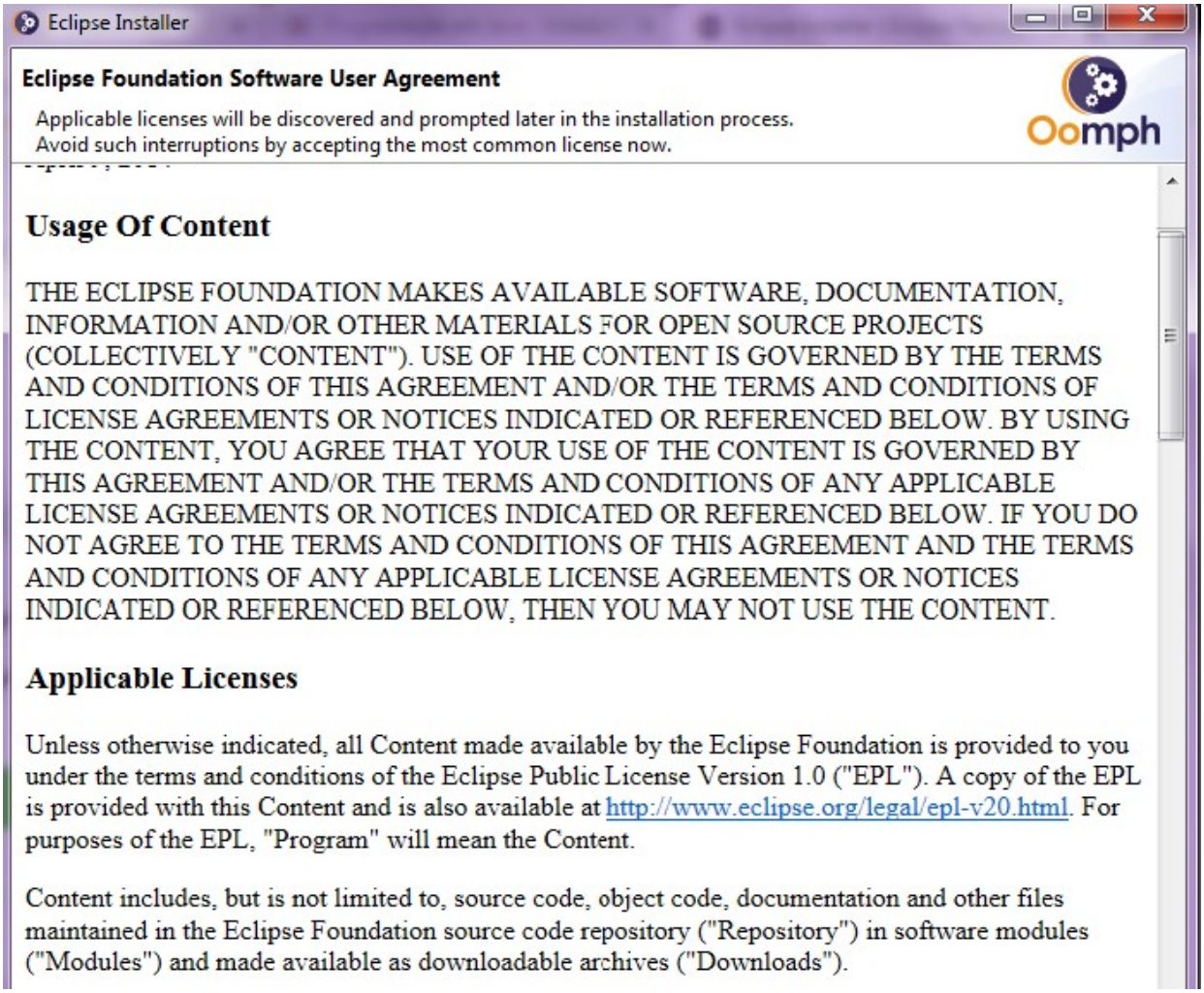


Figura 5. Termos de Uso do Eclipse

Ao aceitar os termos de uso, o eclipse começará a ser instalado, ao final da instalação, clique em *Launch*:

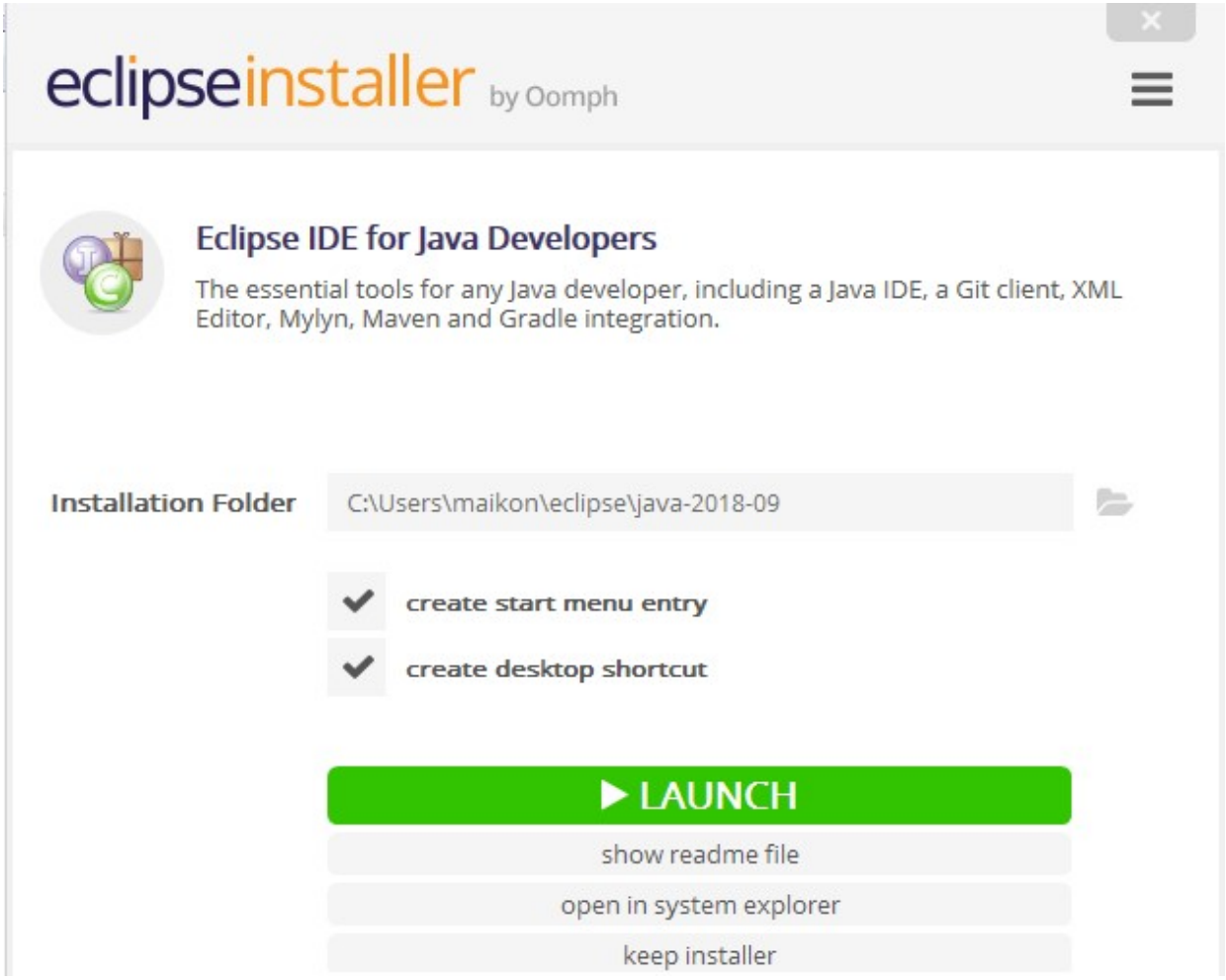


Figura 6. Final da instalação do Eclipse

Depois de carregado, o programa perguntará onde seus arquivos deverão ser armazenados. Por padrão, o Eclipse salva todos os projetos em uma pasta denominada **Workspace**, dentro do seu diretório pessoal, mas você pode alterar tanto o local de destino, quanto o nome da pasta. Você também pode clicar na caixa *"use this as default"* para que a pasta seja padrão para futuros projetos.

Após executar isso basta clicar em *Launch*.

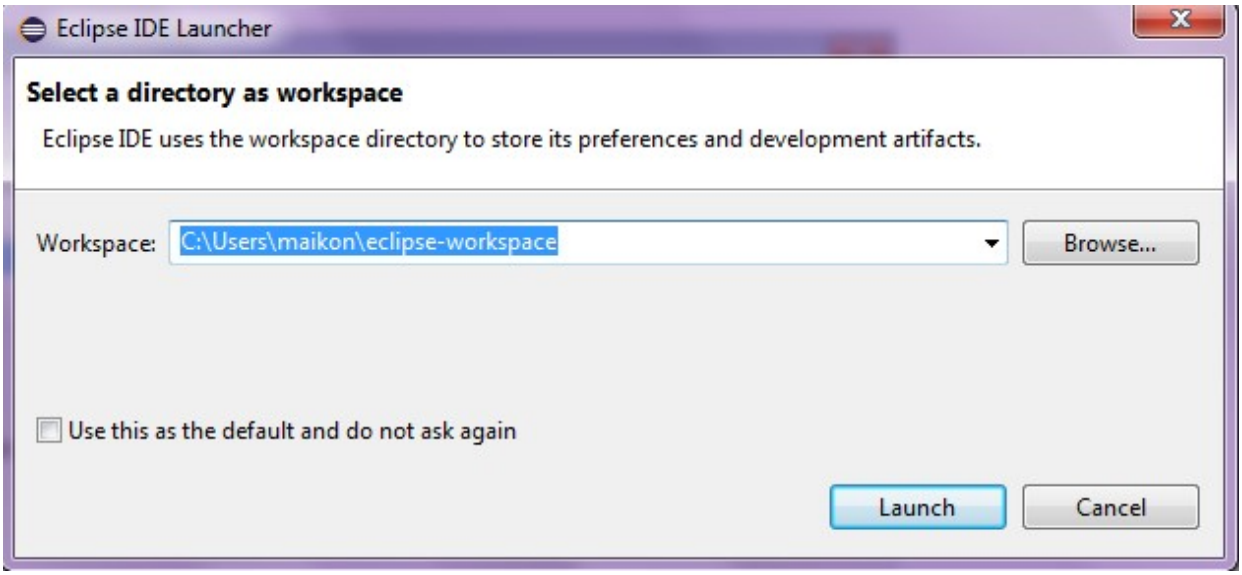


Figura 7. Armazenamento de arquivos

Após isso você verá uma tela de boas vindas do Eclipse. A partir dela você pode acessar

tutoriais, samples, projetos do repositório Git e criar o seu primeiro projeto em Java dentro da IDE. Para isso, é só clicar em *"Create a new Java project"*.

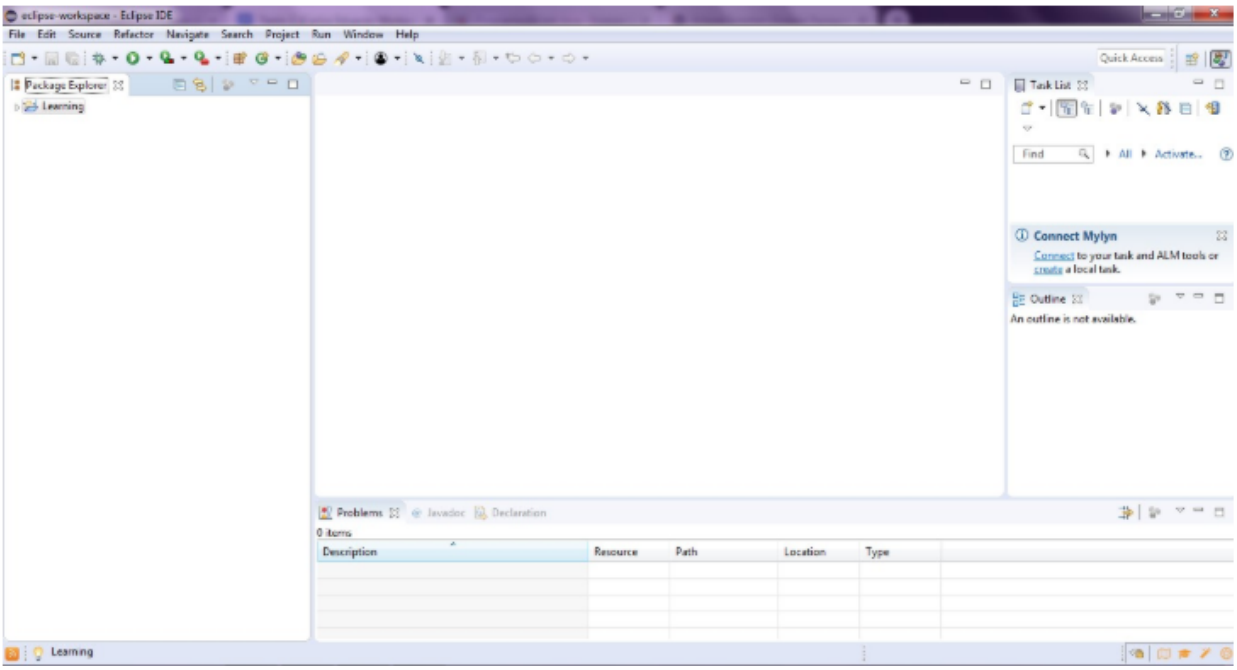


Figura 8. Tela inicial Eclipse

Pronto!!! Eclipse instalado agora você poderá programa em Java.

Entrada e Saída de dados

Basicamente toda a programação consiste em três etapas, a primeira é a **entrada de dados**, onde o sistema irá receber os dados do usuário, a segunda etapa é o **processamento de dados**, que é uma etapa que não é possível prever quando o usuário irá executar mas que basicamente irá processar (manipular) os dados e fazer a sua visualização ou mesmo o seu armazenamento e por fim teremos a terceira etapa, a **saída de dados**, onde poderemos ter diversas formas de retorno de informações para o usuário, como por exemplo, em um monitor, em um display, em um led, em uma impressora e etc.

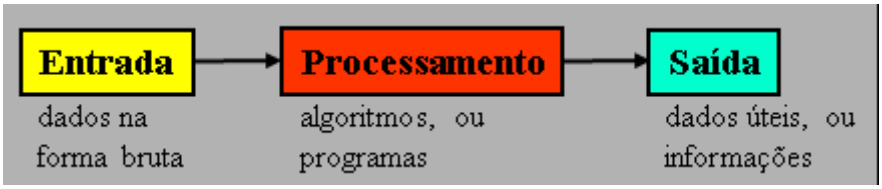


Figura 9. Processamento de dados

Podemos ter uma saída bem simples de dados usando a seguinte programação:

```
class Hello World {
    public static void main(String[] args) {
        System.out.println("Hello World");
```

```
    }
}
```

Em nosso exemplo, nosso programa apenas imprime “Hello World” na tela. Para fazer isso, usamos o método System.out.println. Todotexto (string) em Java é delimitado por aspas duplas, e toda instrução(comando) deve terminar com um ponto e vírgula. Note também que o texto “Hello World” está entre parênteses, que indica o início e término de um parâmetro do método.
System.out.println(“Hello World”);

Classe Scanner

O pacote de classes ***java.util*** disponibilizou a classe Scanner, que implementa operações de entrada de dados pelo teclado.
Alguns métodos da classe Scanner:

String next() - retorna uma cadeia de caracteres simples, ou seja, que não usa o caractere espaço em branco;

- **double nextDouble()** - retorna um número em notação de ponto flutuante normalizada em precisão dupla de 64 bits (usado para receber valores reais ou monetários);
- **boolean hasNextDouble()** - retorna true se o próximo dado de entrada pode ser interpretado como um valor double;
- **int nextInt()** - retorna um número inteiro de 32 bits;
- **boolean hasNextInt()** - retorna true se o próximo dado de entrada pode ser interpretado como um valor int;
- **String nextLine()** - retorna uma cadeia de caracteres, por exemplo: “Generation Desenvolvedor Java”;
- **long nextLong()** - retorna um número inteiro de 64 bits.
- **String next()** - retorna uma cadeia de caracteres simples, ou seja, que não usa o caractere espaço em branco;

Exemplo de um programa utilizando a classe Scanner:

```
import java.util.Scanner;


 public class EntradaSaidaJava {
        public static void main(String[] args) {
```

```
 * 1) entrada de dados
     * 2) processamento de dados
     * 3) saída de dados
    *
        System.out.println("Informe a idade do seu cachorro: ");
            Scanner in = new Scanner( System.in );


            int idadeCachorro = in.nextInt();
            idadeCachorro = idadeCachorro * 7;

        System.out.println("O seu cachorro tem "+idadeCachorro+" anos.");
        }
    }
}
```

Declaração de Variáveis e Constantes

Variável

Para criar um identificador (nome da variável) em Java, precisamos seguir algumas regras, listadas a seguir:

> - Deve conter apenas letras, _ (underline), $ ou os números de 0 a 9
> - Deve obrigatoriamente se iniciar por uma letra (preferencialmente), _ ou $
> - Deve iniciar com uma letra minúscula (boa prática)
> - Não pode conter espaços
> - Não podemos usar palavras-chave da linguagem
> - O nome deve ser único dentro de um escopo

Além disso, o Java é case sensitive, o que significa que os nomes de variáveis diferenciam maiúsculas de minúsculas.

Sintaxe:

tipo identificador [= valor];

onde ***tipo*** é o tipo de dado que a varável irá armazenar, ***identificador*** é seu nome, e ***valor*** é o valor inicial atribuído à variável, o qual é opcional (denotado pelos colchetes, que não devem ser digitados na declaração).

Exemplo de declaração de variáveis:

```
1 int numero;
2 String nome;
```

Figura 10. Declaração de variáveis

Neste caso estamos declarando duma variável chamada *numero* do tipo inteira e uma outra variável chamada *nome* do tipo string.

Atribuição de valores para as variáveis

byte a = 45;
char t = 'T';
int valor = 200;
float x = 98.80;
char sexo = 'F';
int dia; // variável declarada e não inicializada
dia = 20; // variável atribuída agora

Categorias de Variáveis

Existem três categorias de variáveis que podem ser declaradas em Java:

- Locais
- De instância
- De classe

Falaremos sobre esses três tipos de variáveis mais adiante neste documento, quando já tivermos apresentado os conceitos de **orientação a objetos**. Porém, vamos conceituá-las de forma muito concisa a seguir, como uma introdução ao assunto:

Variáveis Locais

Podem ser utilizadas dentro do método onde foram declaradas, não sendo acessíveis de outros pontos do programa.

Variáveis de Instância

Uma classe pode conter variáveis que são declaradas fora dos métodos, chamadas de Variáveis de Instância. São usadas pelos objetos para armazenar seus estados.

Seus valores são específicos de cada instância e não são compartilhados entre as instâncias.

Variáveis de Classe

Variáveis declaradas como estáticas são variáveis compartilhadas entre todos os objetos instanciados a partir de uma classe. Por isso, essas variáveis também são conhecidas como Variáveis de Classe.

Constante

Uma constante é declarada quando precisamos lidar com dados que não devem ser alterados durante a execução do programa. Para isso, utilizamos a palavra reservada ***final*** para que a variável seja inicializada uma única vez.

Exemplos de declaração de constantes:

```
1 final float PI = 3.1416F;
2 final String NOME_PAGINA = "home";
```

Figura 11. Declaração de constantes

Por convenção, usamos letras maiúsculas para declarar constantes e assim distingui-las das variáveis.

Tipos de dados

Tipos Primitivos

Tipo	Descrição	Tamanho (“peso”)
byte	Valor inteiro entre -128 e 127 (inclusivo)	1 byte
short	Valor inteiro entre -32.768 e 32.767 (inclusivo)	2 bytes
int	Valor inteiro entre -2.147.483.648 e 2.147.483.647 (inclusivo)	4 bytes
long	Valor inteiro entre -9.223.372.036.854.775.808 e 9.223.372.036.854.775.807 (inclusivo)	8 bytes
float	Valor com ponto flutuante entre $1,40129846432481707 \times 10^{-45}$ e $3,40282346638528860 \times 10^{38}$ (positivo ou negativo)	4 bytes
double	Valor com ponto flutuante entre $4,94065645841246544 \times 10^{-324}$ e $1,79769313486231570 \times 10^{308}$ (positivo ou negativo)	8 bytes
boolean	true ou false	1 bit
char	Um único caractere Unicode de 16 bits. Valor inteiro e positivo entre 0 (ou ‘\u0000’) e 65.535 (ou ‘\uffff’)	2 bytes

Figura 12. Tipos primitivos de dados

> ***Se liga***
> Nenhum tipo primitivo da linguagem Java permite o armazenamento de texto. O tipo primitivo **char** armazena apenas um caractere. Quando é necessário armazenar um texto, devemos utilizar o tipo **String**. Contudo, é importante salientar que o tipo String não é um tipo primitivo.

Operadores

Para manipular os valores das variáveis de um programa, devemos utilizar os operadores oferecidos pela linguagem de programação adotada. A linguagem Java possui diversos operadores e os principais são categorizados da seguinte forma:

• Aritmético(+, -, *, /, %) • Atribuição(=, +=, -=, *=, /=, %=)
• Relacional(==, !=, <, <=, >, >=)
• Lógico(&&, ||)

Exemplos:

Operadores Aritméticos

```
int umMaisUm = 1 + 1; // umMaisUm = 2
int tresVezesDois = 3 * 2; // tresVezesDois = 6
int quatroDivididoPor2 = 4 / 2; // quatroDivididoPor2 = 2
int seisModuloCinco = 6 % 5; // seisModuloCinco = 1
int x = 7;
```

```
x = x + 1 * 2; // x = 9 7 x = x - 3; // x = 6 8
x = x / (6 - 2 + (3*5)/(16-1)); // x = 2
```

> **Se liga**
> O módulo de um número x, na matemática, é o valor numérico de x desconsiderando o seu sinal (valor absoluto). Na matemática expressamos o módulo da seguinte forma: |−2|=2. Em linguagens de programação, o módulo de um número é o resto da divisão desse número por outro. No exemplo acima, o resto da divisão de 6 por 5 é iguala 1. Além disso, lemos a expressão 6%5 da seguinte forma: seis módulo cinco.

Operadores de Atribuição

```
int valor = 1; // valor = 1
valor += 2; // valor = 3
valor -= 1; // valor = 2
valor *= 6; // valor = 12
valor /= 3; // valor = 4
valor %= 3; // valor = 1
```

Operadores Relacionais

```
int valor = 2;
boolean t = false;
t = (valor == 2); // t = true
t = (valor != 2); // t = false
t = (valor < 2); // t = false
t = (valor <= 2); // t = true
t = (valor > 1); // t = true
t = (valor >= 1); // t = true
```

Operadores Lógicos

```
int valor = 30;
boolean teste = false;
teste = valor < 40 && valor > 20; // teste = true
teste = valor < 40 && valor > 30; // teste = false
teste = valor > 30 || valor > 20; // teste = true
teste = valor > 30 || valor < 20; // teste = false
teste = valor < 50 && valor == 30; // teste = true
```

Palavras reservadas

O java faz uso de algumas palavras que não podem ser utilizadas, por exemplo, para criar nome de variáveis, classes, métodos e etc... A seguir listamos algumas palavras reservadas:

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

Figura 13. Palavras reservadas

Estruturas de Controle if ( )...else

Quando queremos tomar uma decisão em nosso código, ou seja, definir outras opções em Java utilizamos o comando **if (se)**.
A sintaxe do if no Java é a seguinte

```
if (condicaoBooleana)
    { codigo; }
```

Uma condição booleana é qualquer expressão que retorne true ou false. Para isso, você pode usar os operadores <, >, <=, >= e outros.
Além da utilização do if, nós poderemos utilizar a cláusula **else (senão)** para indicar o comportamento que deve ser executado no caso da expressão booleana ser falsa.
Vejamos o exemplo abaixo para identificar essas duas cláusulas:

```
Scanner entrada = new Scanner(System.in);

System.out.print("Nome: ");
String nome = entrada.nextLine();
System.out.print("Peso: ");
int peso = entrada.nextInt();
System.out.print("Altura: ");
double altura = entrada.nextDouble();
double imc = peso / (altura * altura);
```

```
if (imc < 18.5)
            System.out.println("Abaixo do peso ideal.");
else    if (imc < 25)
            System.out.println("Peso ideal.");
else    if (imc < 30)
            System.out.println("Acima do peso.");
else    if (imc < 35)
            System.out.println("Obesidade grau I.");
else    if (imc < 40)
            System.out.println("Obesidade grau II.");
else
            System.out.println("Obesidade grau III.");
            System.out.println("Muito cuidado com seu peso.");
```

Em relação a este exemplo caso o seu IMC for maior ou igual a 40 irá aparecer duas mensagens (*"Obesidade grau III" e "Muito cuidado com o seu peso"*) atendendo a nossa necessidade.
Mas caso o seu IMC for 34 além de apresentar a mensagem "*Obesidade grau I*" irá aparecer "*Muito cuidado com o seu peso*" não atendendo a nossa necessidade.
Isso se deve ao fato de termos uma regra em relação as chaves de demarcação de cada estrutura if( ). Só se torna obrigatório o uso das chaves { } quando tiver mais de uma linha de execução no bloco, quando tiver apenas uma única linha de execução não se torna necessário.
Nesse mesmo exemplo se quisermos colocar mais de uma opção dentro da condição iremos utilizar os operadores lógicos, como no exemplo abaixo:

```
if (sexo == 'F' && imc < 19.1) {
System.out.println("Abaixo do peso."); }
else if (sexo == 'F' && imc <= 25.8) {
System.out.println("Peso ideal."); }
else if (sexo == 'F' && imc <= 27.3) {
System.out.println("Um pouco acima do peso."); }
else if (sexo == 'F' && imc <= 32.3) {
System.out.println("Acima do peso ideal."); }
else if (sexo == 'F') { System.out.println("Obeso."); }
else if (sexo == 'M' && imc < 20.7) {
System.out.println("Abaixo do peso."); }
else if (sexo == 'M' && imc <= 26.4) {
System.out.println("Peso ideal."); }
else if (sexo == 'M' && imc <= 27.8) {
System.out.println("Um pouco acima do peso."); }
else if (sexo == 'M' && imc <= 31.1) {
System.out.println("Acima do peso ideal."); }
```

```
else if (sexo == 'M') {
System.out.println("Obeso."); }
```

O operador lógico && avalia as expressões do lado esquerdo e direito e retorna apenas um resultado booleano. Para a expressão completa ser verdadeira, tanto o lado direito como o lado esquerdo devem ser verdadeiras, mas para que a expressão completa seja falsa, pelo menos um lado deve ser falso.

Ainda existe o operador lógico “OU”, representado por ||. Podemos ainda mudar o código-fonte de nosso exemplo para usá-lo.

```
if ((sexo == 'F' && imc < 19.1) || (sexo == 'M' && imc < 20.7)) { System.ou
else if ((sexo == 'F' && imc <= 25.8) || (sexo == 'M' && imc <= 26.4)) {
System.out.println("Peso ideal."); }
else if ((sexo == 'F' && imc <= 27.3) || (sexo == 'M' && imc <= 27.8)) {
System.out.println("Um pouco acima do peso."); }
else if ((sexo == 'F' && imc <= 32.3) || (sexo == 'M' && imc <= 31.1)) {
System.out.println("Acima do peso ideal."); }
else { System.out.println("Obeso."); }
```

Veja que, usando o operador ||, conseguimos diminuir bastante a quantidade de linhas de programação, porém deixamos as expressões booleanas dos ifs mais complexas, pois agrupamos duas expressões que usam o operador && dentro de outra que usa o operador ||.
O operador lógico || avalia as expressões dos dois lados e retorna um único resultado booleano. Para que a expressão completa seja verdadeira, pelo menos um lado deve ser verdadeiro.

Estrutura de Controle switch( )

Sintaxe:

```
switch (Expressao){
case valor1: conjuntoDeSentencas;
break;
case valor2: SentencasAlternativas;
break;
case valor3: SentencasAlternativas2;
break;
case valor4: SentencasAlternativas3;
break;
}
```

A estrutura de controle switch pode receber um valor do tipo primitivo byte, short, char e int, String e outros tipos wrappers que envolvem tipos primitivos, como o Character, Byte, ShorteInteger.
O comando switch pode ter vários possíveis caminhos de decisão (casos). O primeiro caso que estiver de acordo com o valor passado para o switch inicia a execução das instruções do caso.
A sentença 'break' atrás de cada opção de case serve para que não avalie o resto de opções e sim que saia diretamente do 'Switch', por isso, dependendo do que quiser fazer, você colocará ou não.
Vamos verificar um exemplo para demonstrar a execução desse laço:

```
switch (i) {
case '1': System.out.println( "i contem um 1");
case '2': System.out.println( "i contem um 2");
case '3': System.out.println( "i contem um 3");
}
```

Exemplo usando o break:

```
Scanner leitor  =  new  Scanner(System.in);
int  dia  =  0;
System.out.println("Digite um numero para encontrar o seu dia: ");
dia  =  leitor.nextInt();
switch  (dia)  {
case  1:
System.out.println("Domingo");
break;
case  2:
System.out.println("Segunda");
break;
case  3:
System.out.println("Terça");
break;
case  4:
System.out.println("Quarta");
break;
case  5:
System.out.println("Quinta");
break;
case  6:
System.out.println("Sexta");
break;
case  7:
System.out.println("Sabado");
break;
```

```
default:
System.out.println("Número digitado é inválido!!!");
break;
}
}
```

Estruturas de Repetição

Essas estruturas fazem com que uma instrução, ou bloco execute repetidamente, enquanto uma expressão seja verdadeira.
Existem 2 tipos:
● Estruturas de repetição incondicional (simples): Repete um número específico de vezes. Estrutura *for*
● Estruturas de repetição condicional: São estruturas de repetição que o controle é feito pela avaliação de expressões condicionais. Ou seja, o número de repetições é indeterminado na fase de programação, será conhecido durante a execução. Estruturas *while* e *do – while*

Estrutura for ( )

```
for (inicialização; condição de execução; Incremento/decremento)
Bloco de Instruções;
```

● Inicialização: É usado para dar valor inicial a variável de controle (contador).
● Condição de execução: É uma expressão lógica que determina a execução associada ao for, geralmente utilizando a variável de controle.
● Incremento/Decremento: Determina como a variável de controle (ou outras variáveis também) será alterada a cada iteração do for.
● A inicialização é feita apenas antes da primeira iteração.
● A execução é encerrada quando a condição de execução for avaliada como falsa.
Exemplo:
Programa que soma os n primeiros números que entraram como argumento.

```
public class ExemploFor {
    public static void main(String[] args) {
    int n=Integer.parseInt(args[0]);
    int i, soma; //(inicialização; condição; incremento)
    for(i=0, soma=0; i<=n; i++) {
        System.out.print(“Soma = ”+soma+” + ”+i);
```

```
        soma+=i;
        System.out.println(“ = ”+soma+”\n”); } } }
```

Estrutura while ( )

O while é uma das formas de fazer loops (laços) em Java. Loops são usados para executar um bloco de código diversas vezes, dependendo de uma condição ser verdadeira.
O exemplo abaixo imprime todos os números em um intervalo informado pelo usuário.

```
Scanner entrada = new Scanner(System.in);

System.out.print("Digite o numero inicial: ");
int numeroInicial = entrada.nextInt();
System.out.print("Digite o numero final: ");
int numeroFinal = entrada.nextInt();
int numeroAtual = numeroInicial;
while (numeroAtual <= numeroFinal) {

System.out.println(numeroAtual); numeroAtual++; }
```

O bloco de código dentro do while será executado enquanto o valor da variável numeroAtual for menor ou igual ao valor da variável numeroFinal. Veja que usamos o operador de incremento para adicionar à variável numeroAtual uma unidade a cada execução do bloco de código do loop.

Estrutura do...while( )

A declaração do-while é similar ao while. Ele executa a instrução pelo menos uma vez e continua executando enquanto a expressão booleana for verdadeira.
Vejamos o exemplo abaixo:

```
public class ExemploDoWhile{
public static void main(String[] args) {
int x=0;
// O do-while garante que pelo menos
//uma vez o codigo seja executado
do {
System.out.println(x);
x++;
} while(x < 10);
```

```
}
}
```

Uma estrutura dentro da outra

Um bloco também pode ser declarado dentro de outro. Isto é, um if dentro de um for, ou um for dentro de um for, algo como:

```
while (condicao) {
     for (int i = 0; i < 10; i++) {
     // código }
}
```

Arrays

Quando desejamos armazenar uma grande quantidade de valores de um determinado tipo, podemos utilizar *arrays*. Um array é um objeto que pode armazenar muitos valores de um determinado tipo.
Podemos imaginar um array como sendo um armário com um determinado número de gavetas. E cada gaveta possui um rótulo com um número de identificação.

Figura 1. Analogia de array

Para declarar um array devemos seguir a seguinte sintaxe:

tipo[] nome_do_array = new tipo[numero_de_elementos];
Ou:
tipo[] nome_do_array = { valor1, valor2, ...,valorx};

Por exemplo, se em uma sala de aula tiver 20 alunos e você quiser declarar 20 variáveis do tipo float usando array, ficaria assim:

float[] nota = new int[20];
float[] nota = {1,2,3,4,5,6,7,8,90.1,10.02,-11,12.9,13.7,14.5,15.7,16.0,17.5,19.3,20.2};

E o nome dos alunos, armazenaríamos em Strings:
String[] nome = new String[20];

Se quisermos acessar os elementos de um array podemos fazer o seguinte:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo
```

E neste mesmo exemplo se quisermos alterar um elemento do array basta fazer o seguinte:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
```

Se quisermos saber o tamanho de um array iremos utilizar a propriedade **length ( )**.

```
public class TamanhoArray {
public` `static` `void` `main(String[] args) {
int[] arrayUm ={12, 3, 5, 68, 9, 6, 73, 44, 456, 65, 321} ;
int[] arrayDois = {43, 42, 4, 8, 55, 21, 2, 45} ;

if (arrayDois.length >8) {
System.out.println ("Tamanho do ArrayDois - Maior que 8!");
} else {
System.out.println("Tamanho do ArrayDois - Menor que 8!"``);
}
System.out.println("\nTamanho do ArrayUm = "+arrayUm.length);
}
}
```

Para percorrer um array podemos utilizar um for-each:

```
String[] cars = {"Volvo", "", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

O exemplo acima pode ser lido assim: **para cada** `String` elemento (chamado **i** - como em **i** NDEX) em **carros** , imprimir o valor de **i** .

Se você comparar o `for` loop e o loop **for-each** , verá que o método for **-each** é mais fácil de escrever, não requer um contador (usando a propriedade length) e é mais legível.

Arrays Multidimensionais

Os **arrays bidimensionais** precisam de dois índices para identificar um elemento particular.

Por exemplo, quando um array é identificado dessa forma "**numero[indiceA][indiceB]**", a variável **numero** é o array, o **indiceA** é a linha e o **indiceB** é identificado como a coluna, fazendo uma identificação de cada elemento no array por número de linha e coluna.

```
public class Inicializando_Arrays_Bidimensionais {

public static void main(String[] args) {

int [][] array1 = { {1, 2, 3}, {4, 5, 6} };
int [][] array2 = { {1, 2}, {3}, {4, 5, 6} };

System.out.println("Valores no array1 passados na linha são");
outputArray( array1 ); //exibe o array 2 por linha
System.out.println("Valores no array2 passados na linha são");
outputArray( array2 ); //exibe o array 2 por linha
}
//FAZ UM LOOP PELAS LINHAS DO ARRAY

public static void outputArray(int [][] array)
{
//FAZ UM LOOP PELAS COLUNAS DA LINHA ATUAL
for( int linha =0; linha < array.length; linha++)
{
//FAZ LOOP PELAS COLUNAS DA LINHA ATUAL
for (int coluna = 0; coluna < array[linha].length; coluna++)
System.out.printf("%d ", array[linha][coluna]);
System.out.println();
}
}
}
```

Funções em java

Função é algo que deve ser feito uma ou várias vezes, sempre que for necessário para se obter um resultado.
– Pode, ou não, receber parâmetros;
– Sempre retorna um resultado;
– Exemplos:

• Mostrar um menu e retornar a opção;
• Solicitar uma entrada ao usuário;
• Efetuar uma operação matemática;
• Ler dados de um arquivo;
A seguir podemos visualizar um exemplo de uma função chamada *MostraMenu* que está sendo chamada pelo *main*.

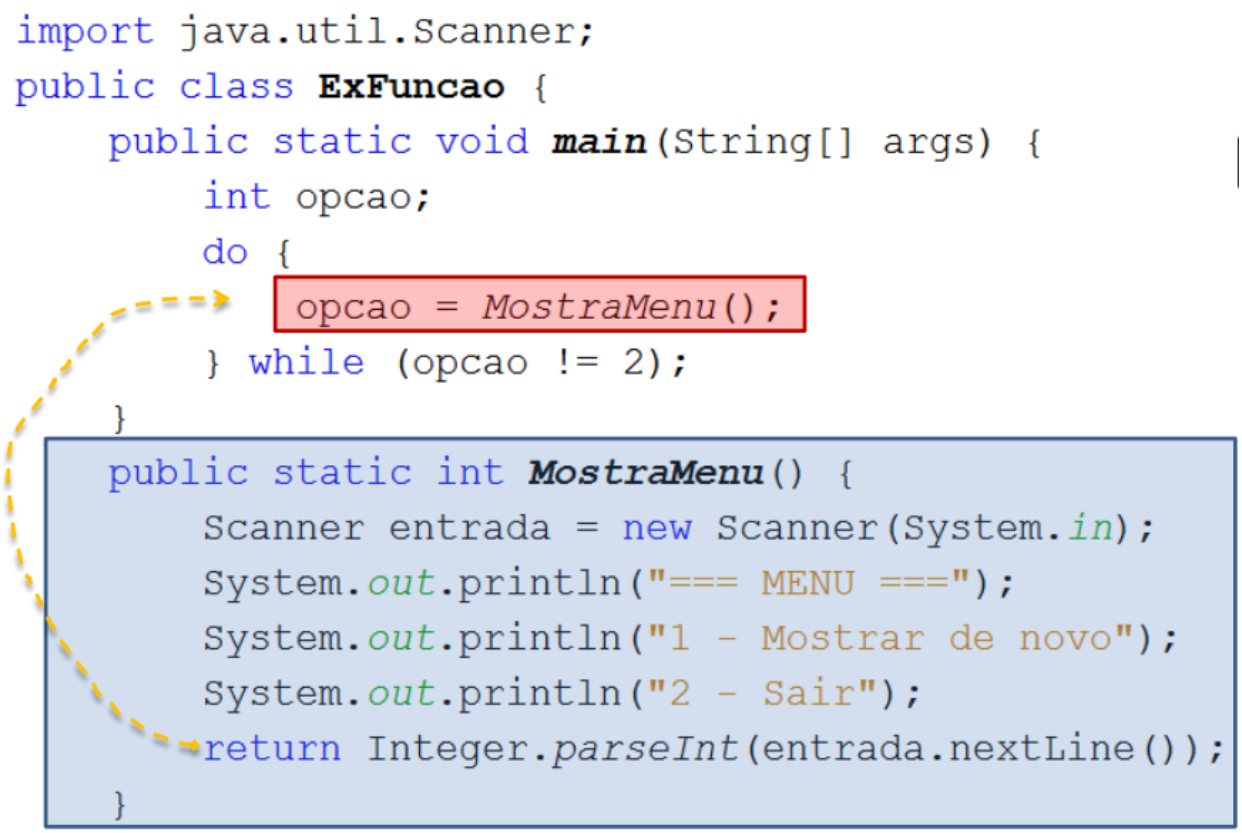


Figura 154. Função em java

Outro exemplo que podemos utilizar uma função é calcular os descontos percentuais em valores de produtos.
– Precisa de quais dados?

• Precisa do valor atual do produto – **double**;
• Precisa do percentual de desconto – **double**;

– Resulta em algum novo dado?
• Sim, é uma função! O valor com desconto – **double**;

```
public static double calcDesc(double va, double pc) {
double vd = va * (pc / 100); return va - vd;
}
```

Orientação a Objetos

Programação Orientada a Objetos (POO) é um paradigma de programação que ajuda a definir a estrutura de programas de computadores, baseado nos conceitos do mundo real, sejam eles reais ou abstratos. A ideia é simular as coisas que existem e acontecem no mundo real no mundo virtual.

Classes e Objetos

Em Java os programas são escritos em pequenos pedaços separados, chamados de **objetos**. Objetos são pequenos programas que guardam dentro de si os dados – em suma, as **variáveis** – que precisam para executar suas tarefas. Os objetos também trazem em si, como sub-rotinas, as instruções para processar esses dados. As variáveis que um objeto guarda são chamadas de **atributos**, e as suas sub-rotinas são chamadas de **métodos**.

Vamos colocar um exemplo bem simples para entender esses conceitos. Pensem em um carro, este carro possui um motor, uma cor, portas, câmbio, etc. Ele também possui comportamentos que, provavelmente, foram o motivo de sua compra, como acelerar, desacelerar, acender os faróis, buzinar e tocar música. Podemos dizer que o carro novo é um *objeto*, onde suas características são seus *atributos* (dados atrelados ao objeto) e seus comportamentos são ações ou *métodos*.
Ao mesmo tempo o seu carro, é apenas um dentre muitos outros dentro da loja, ou seja, seu carro pode ser classificado como um carro que o seu carro será uma *instância* dessa classe chamada carro.

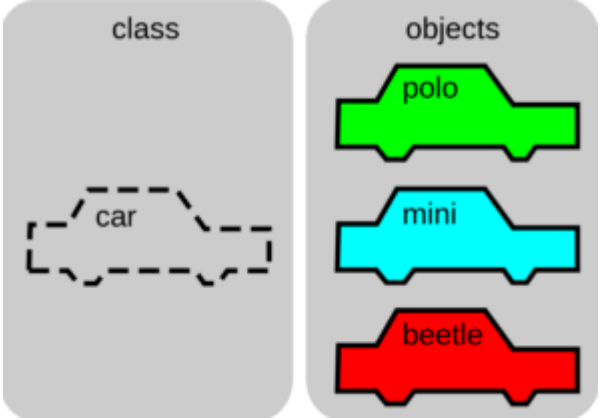


Figura 1. Relação classe e objetos

Podemos dar um outro exemplo em um domínio bancário:

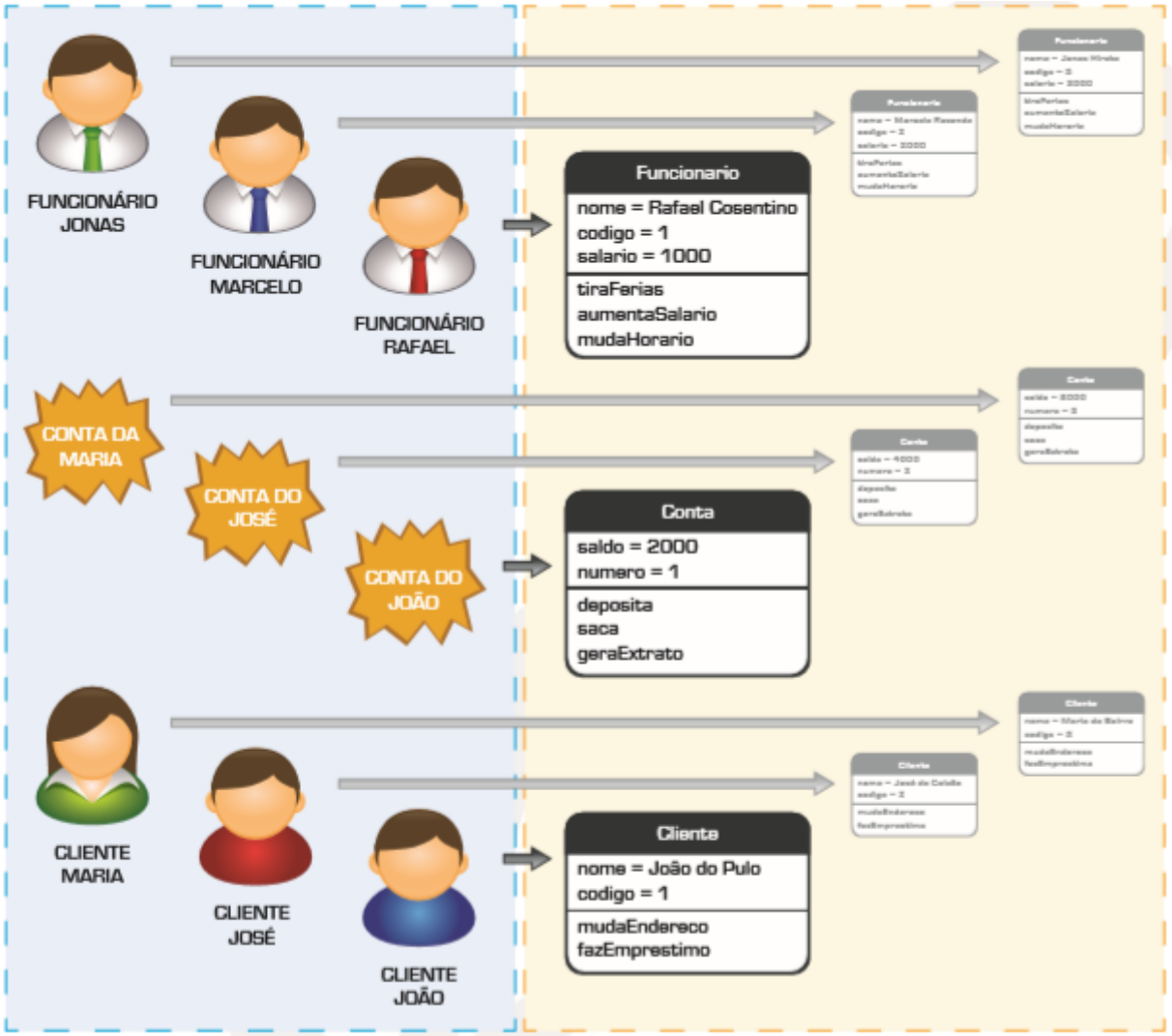


Figura 1. Domínio bancário

Podemos representar uma classe através de diagramas UML.

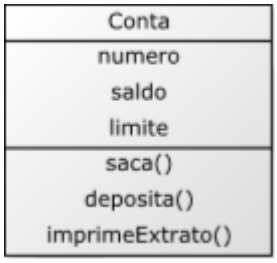


Figura . Diagrama UML da classe Conta

Repare que a classe em si é um conceito abstrato, como um molde, que se torna concreto e palpável através da criação de um objeto. Chamamos essa criação de *instanciação da classe*, como se estivéssemos usando esse molde (classe) para criar um objeto.

Bem, visto os conceitos fundamentais de orientação à objetos, agora vamos aplicar todos esses conceitos na prática utilizando a linguagem de programação Java.

Colocamos uma classe genérica *Pessoa*, onde teria que guardar os vários pedaços do seu nome, então para isso teríamos que criar alguns atributos para a mesma, por exemplo:

```
public class Pessoa {

private String primeiroNome;

private String ultimoNome;

private String nomesDoMeio;

}
```

Criamos a classe Pessoa com três atributos, que será modelo para a criação de outros objetos. Explicando melhor, então criamos três atributos do tipo **private** que serve para deixar o atributo privado, ou seja, somente métodos da própria classe Pessoa que poderá acessá-lo e manipular o mesmo.
Mas não basta criar os atributos, temos que também dar algumas funcionalidades para os mesmos, para isso criamos um método para retornar o nome completo por exemplo.

```
public String getNomeCompleto() {

      String nomeCompleto = primeiroNome + " " + nomesDoMeio + " " + ultimoNome;
      return nomeCompleto;
  }
```

Figura . Criação de método
A primeira linha, public String getNomeCompleto(), especifica o método. Primeiro, declara-se, através da palavra-chave public, que o método é público – isto é, qualquer método, de qualquer classe, pode invocá-lo. O método retorna objetos do tipo String e se chama getNomeCompleto. O par de parênteses vazio significa que ele não recebe parâmetro algum.

O conteúdo do método vem entre um par de chaves. Dentro do método, declara-se uma nova variável, do tipo String, chamada nomeCompleto. Ao contrário de variáveis como primeiroNome, nomeCompleto não é um atributo, mas sim uma *variável local*. primeiroNome existirá desde a criação do objeto até sua retirada da memória, mas nomeCompleto só existirá enquanto o método getNomeCompleto() estiver sendo executado, e para cada chamada do método uma nova variável será criada.

A variável nomeCompleto recebe o resultado da concatenação de String. O sinal de atribuição é =, e a concatenação de String é feita através do operador +. Ao final, um ponto-e-vírgula sinaliza o fim deste comando. Abaixo, temos o comando return. Quando ele é invocado, o método termina e o valor que está à sua frente (no caso, o valor referenciado pela variável nnomeCompleto) é retornado.
Até agora ainda não construímos nenhum objeto, para isto Java tem uma ferramenta

chamada *construtor*. Geralmente os construtores tem o mesmo nome da classe. Esse construtor cria um novo objeto e este novo objeto é armazenado na variável *pessoa*.

```
Pessoa pessoa = new Pessoa();
```

Agora vamos criar uma pessoa chamada Francisco Pinho Nunes.

```
public class ProgramaNome {
public static void main(String[] args) {
Pessoa pessoa = new Pessoa();
pessoa.primeiroNome = “Francisco”;
pessoa.nomeDoMeio = “Pinho”;
pessoa.ultimoNome = “Nunes”;
System.out.println(pessoa.getNomeCompleto());
}
}
```

No entanto, isto não é possível porque os atributos são privados. Apenas os métodos da classe *Pessoa* podem acessá-los. Isso pode ser solucionado de várias maneiras, e uma das mais elegantes é criando o nosso próprio construtor, como abaixo:

```
public Pessoa(String primeiro, String meio, String ultimo) {

      primeiroNome = primeiro;
      ultimoNome = ultimo;
      nomesDoMeio = meio;
  }
```

Figura 2018. Criando um construtor

A declaração do construtor é sempre o nome da classe seguido pela lista de parâmetros. A palavra *public* indica que o construtor é público, de modo que pode ser invocado por qualquer classe. Um ponto importante sobre construtores é que eles não criam nem retornam objetos; quem faz isso é a palavra reservada *new*. O construtor apenas executa algum procedimento sobre o objeto criado pelo comando *new*. Este construtor, no caso, recebe os nomes como parâmetros e os atribui aos atributos.

Agora, podemos trocar as quatro primeiras linhas do método *main()* inválido por apenas a seguinte:

```
Pessoa pessoa = new Pessoa( "Francisco", "Pinho", "Nunes" );
```

Só para identificar, vejamos como ficou a classe *Pessoa* completa:

```
public class Pessoa {
    private String primeiroNome;
    private String ultimoNome;
    private String nomesDoMeio;


    public Pessoa(String primeiro, String meio, String ultimo) {
      primeiroNome = primeiro;
      ultimoNome = ultimo;
      nomesDoMeio = meio;
    }


    public String getNomeCompleto() {
      String nomeCompleto = primeiroNome + " " + nomesDoMeio + " " + ultimoNome;
      return nomeCompleto;
    }

  }
```

Figura 2. Classe Pessoa completa

E a classe ProgramaNome, que utiliza a classe Pessoa para gerar um nome completo a partir das partes.

```
public class ProgramaNome {

public static void main(String[] args) {

Pessoa pessoa = new Pessoa("Francisco","Pinho","Nunes");

System.out.println(pessoa.getNomeCompleto());
} }
```

Objeto this

A palavra reservada *this* faz referência ao próprio objeto, quando usado dentro de um método, por exemplo.
No código-fonte da classe Aeronave, vamos criar um método chamado alterarTotalAssentos, que receberá um argumento com o novo número de passageiros a ser atribuído à variável de instância totalAssentos.

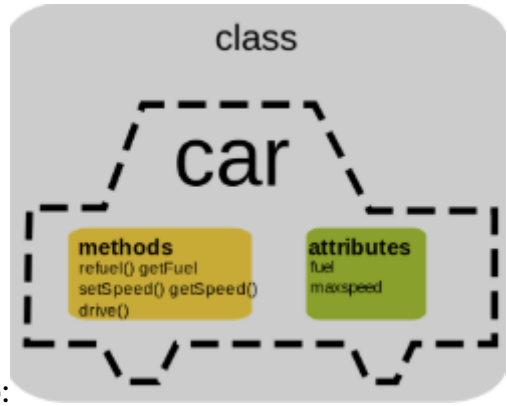
```
class Aeronave {
int totalAssentos;
int assentosReservados;
```

```
void reservarAssentos(int assentos) {
assentosReservados += assentos;
}

int calcularAssentosDisponiveis() {
return totalAssentos - assentosReservados;
}
void alterarTotalAssentos(int totalAssentos) {
this.totalAssentos = totalAssentos;
}
```

Estamos dizendo que queremos atribuir o valor da variável local à variável de instância. Quando usamos *this*, estamos deixando essa informação explícita.

Encapsulamento



Vamos voltar ao exemplo que demos do carro:
Figura 220. Classe Carro
Os métodos do carro, como acelerar, podem usar atributos e outros métodos do carro como o tanque de gasolina e o mecanismo de injeção de combustível, respectivamente, uma vez que acelerar gasta combustível.
Mas alguns métodos e atributos não podem ser permitidos qualquer tipo de alteração, ou seja, eles não são visíveis para fora do carro.
Na POO, um atributo ou método que não é visível de fora do próprio objeto é chamado de "*privado*" e quando é visível, é chamado de "*público*".

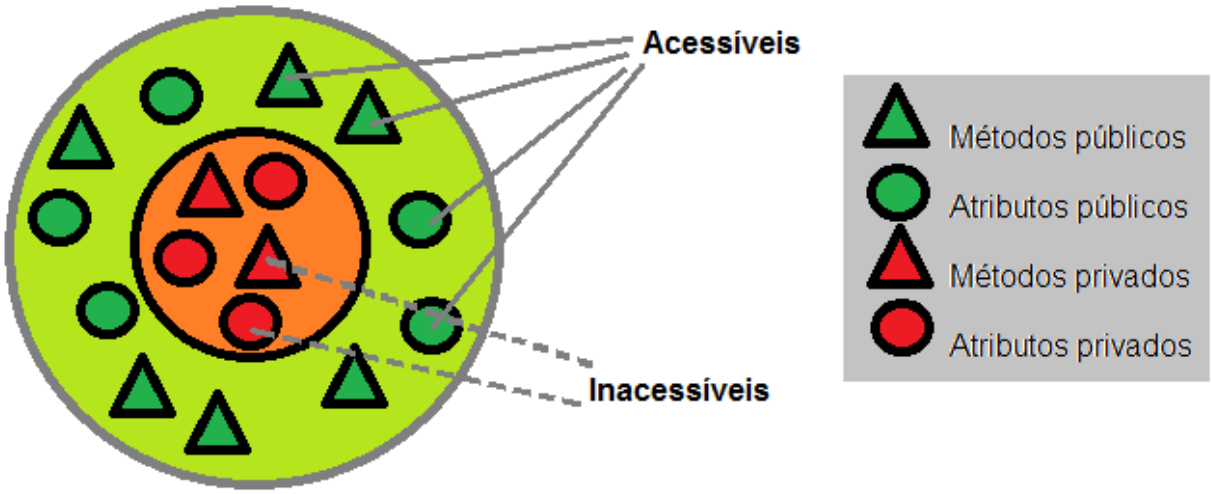


Figura 2. Métodos/Atributos públicos e privados

Ler ou alterar um atributo encapsulado pode ser feito a partir de *getters e setters* (colocar referência).
Esse *encapsulamento* de atributos e métodos impede o chamado *vazamento de escopo*, onde um atributo ou método é visível por alguém que não deveria vê-lo, como outro objeto ou classe. Isso evita a confusão do uso de variáveis globais no programa, deixando mais fácil de identificar em qual estado cada variável vai estar a cada momento do programa, já que a restrição de acesso nos permite identificar quem consegue modificá-la.
Exemplo:

```
public class Carro {
private Double velocidade;
private String modelo;
private MecanismoAceleracao mecanismoAceleracao;
private String cor;

/* Repare que o mecanismo de aceleração é inserido no carro ao ser construí

public Carro(String modelo, MecanismoAceleracao mecanismoAceleracao) {
this.modelo = modelo;
this.mecanismoAceleracao = mecanismoAceleracao; this.velocidade = 0; }

public void acelerar() {
 this.mecanismoAceleracao.acelerar(); }

public void frear() { /* código do carro para frear */ }

public void acenderFarol() { /* código do carro para acender o farol */ }
public Double getVelocidade() { return this.velocidade }

private void setVelocidade() {
```

```
/* código para alterar a velocidade do carro */ /* Como só o próprio carro
}

public String getModelo() { return this.modelo; }

public String getCor() { return this.cor; }

/* podemos mudar a cor do carro quando quisermos */

public void setCor(String cor) { this.cor = cor; }
}
```

Herança

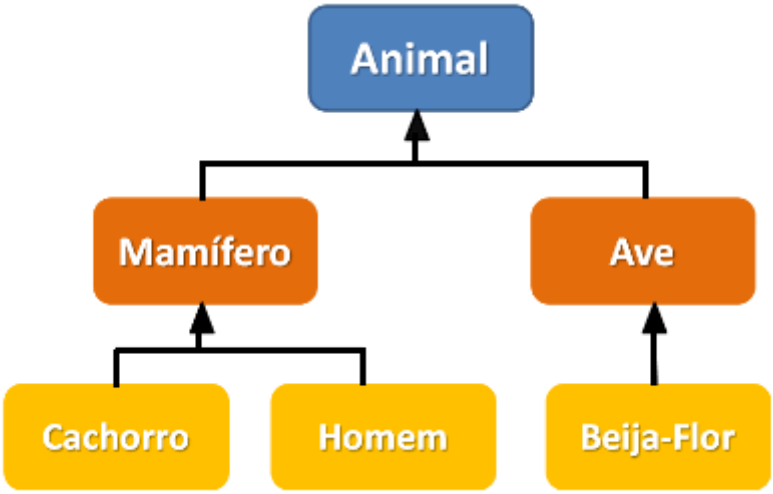


Figura 4. Representação da herança

Quando dizemos que uma classe A *é um tipo de* classe B, dizemos que a classe A *herda* as características da classe B e que a classe B é *mãe* da classe A, estabelecendo então uma relação de **herança** entre elas.
Neste caso, por exemplo, o cachorro, herdará todas as características do mamífero e o mamífero herda as características de um animal.
Por exemplo, poderíamos ter uma classe Animal que representasse animais em geral e as atividades que todos eles executam. Uma dessas atividades é comer, e nossa classe poderia ser como a abaixo.

```
public class Animal {

public void comer(String alimento) {

System.out.println("Eu estou comendo "+alimento);
}
}
```

Uma classe de mamíferos *estenderia* a classe de animais, porque mamíferos podem fazer coisas que outros animais não podem – por exemplo, gerar leite. Uma classe Mamifero estenderia a classe *Animal* usando a palavra reservada **extends** em sua declaração:

```
 public class Mamifero extends Animal {
    }
```

Um novo método, *lactar()*, também poderia ser adicionado:

```
public String lactar() {
return  "leite" ;
}
```

Como mamíferos são animais, também precisam do método *comer()*, e aí entra uma das vantagens da herança: como a classe Mamifero estende a classe Animal, ela *herda* todos os métodos de Animal, então não é preciso reescrevê-los.

```
public class AnimaisAcao {

public static void main(String[] args) {

Animal animal = new Animal();`

Mamifero mamifero = new Mamifero();

animal.comer (" plantas ");

mamifero.comer (" queijo ");

String produto = mamifero.lactar();

System.out.println("O produto da lactação é "+ produto);
}
}
```

O programa **acima** representa bem isso: note como o método comer() é invocado, mesmo sem ser declarado em Mamifero.

A figura abaixo mostra a impressão deste programa:

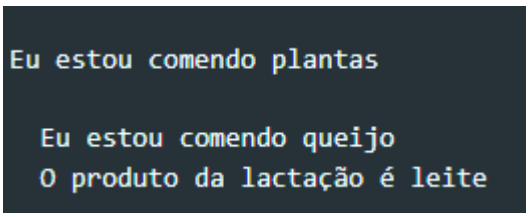


Figura 25. Resultado da execução

Sobrescrita de métodos

Uma classe pode ter dois ou mais métodos com o mesmo nome, desde que os tipos de seus argumentos sejam distintos.
Isso é útil quando queremos implementar um método em função de outro.
Então falamos que sobrecarga de métodos é um conceito simples da orientação a objetos, que permite a criação de vários métodos com o mesmo nome, mas com parâmetros diferentes.
Para exemplifica vamos pegar uma classe Aeronave que irá permitir reservar os assentos normais e os especiais.

```
class Aeronave {
int totalAssentosNormais;
int totalAssentosEspeciais;
int assentosNormaisReservados;
int assentosEspeciaisReservados;

void reservarAssentos(int assentos) { this.assentosNormaisReservados += ass
}

int calcularAssentosDisponiveis() {
return totalAssentosNormais - assentosNormaisReservados + totalAssentosEspe
}
```

Agora vamos sobrecarregar o método *reservarAssentos*, incluindo uma nova versão com parâmetros adicionais:

```
void reservarAssentos(int assentosNormais, int assentosEspeciais) {

this.assentosNormaisReservados += assentosNormais;

this.assentosEspeciaisReservados += assentosEspeciais;
}
```

O método acima recebe, além do número de assentos normais, o total de assentos especiais a serem reservados. Isso é sobrecarga de métodos! Temos duas versões de métodos com o nome *reservarAssentos*. Vamos ver as assinaturas desses métodos?

```
void reservarAssentos(int)

void reservarAssentos(int, int)
```

Sobrecarga de métodos é algo simples de ser feito, mas tem uma restrição que a própria linguagem impõe. Não é possível ter duas versões de métodos com a mesma assinatura. Por exemplo, seria impossível ter o método a seguir na *classeAeronave*:

```
void reservarAssentos(int assentosEspeciais) {

this.assentosEspeciaisReservados += assentosEspeciais;
}
```

O código acima seria uma tentativa de criar uma versão do método *reservarAssentos*, para reservar assentos especiais, mas não funcionaria (nem compilaria), porque a classe Aeronave já possui um método *reservarAssentos* que recebe um int.

Interface

Quando duas (ou mais) classes possuem comportamentos comuns que podem ser separados em uma outra classe, dizemos que a “classe comum” é uma *interface*, que pode ser “herdada” pelas outras classes. Note que colocamos a interface como “classe comum”, que pode ser “herdada” (com aspas), porque uma interface não é exatamente um classe, mas sim um conjunto de métodos que todas as classes que herdarem dela devem possuir (implementar) - portanto, uma interface não é “herdada” por uma classe, mas sim *implementada*. No mundo do desenvolvimento de software, dizemos que uma interface é um “contrato”: uma classe que implementa uma interface deve fornecer uma implementação a **todos** os métodos que a interface define, e em compensação, a classe implementadora pode dizer que ela é do tipo da interface.

Abaixo é possível ver um exemplo de uma interface chamada *FiguraGeometrica* com três assinaturas de métodos que virão a ser implementados pelas classes referentes às figuras geométricas.

```
public interface FiguraGeometrica
{
 public String getNomeFigura();
 public int getArea();
 public int getPerimetro();
}
```

Figura 2. Interface FiguraGeometrica

Para realizar a chamada/referência a uma interface por uma determinada classe, é necessário adicionar a palavra-chave implements ao final da assinatura da classe que irá implementar a interface escolhida.

Sintaxe:
public class nome_classe implements nome_interface

Onde:

- nome_classe – Nome da classe a ser implementada.
- nome_Interface – Nome da interface a se implementada pela classe.

Abaixo é possível ver duas classes que implementam a interface FiguraGeometrica, uma chamada Quadrado e outra Triangulo.

Classe Quadrado

```
/*
@author Robson Fernando Gomes
*/
public class Quadrado implements FiguraGeometrica {
private int lado;

public int getLado() {
return lado;
}

public void setLado(int lado) {
this.lado = lado;
}

`@Override`
public int getArea() {
int area = 0;
area = lado * lado;
return area;
}
```

```
@Override
public int getPerimetro() {
int perimetro = 0;
perimetro = lado * 4;
return perimetro;
}

@Override
public String getNomeFigura() {
return "quadrado";
}
}
```

Classe Triangulo

```
public class Triangulo implements FiguraGeometrica {

private int base;
private int altura;
private int ladoA;
private int ladoB;
private int ladoC;

public int getAltura() {
return altura;
}

public void setAltura(int altura) {
this.altura = altura;
}

public int getBase() {
return base;
}

public void setBase(int base) {
this.base = base;
}

public int getLadoA() {
return ladoA;
}

public void setLadoA(int ladoA) {
this.ladoA = ladoA;
}
```

```
public int getLadoB() {
return ladoB;
}

public void setLadoB(int ladoB) {
this.ladoB = ladoB;
}

public int getLadoC() {
return ladoC;
}

public void setLadoC(int ladoC) {
this.ladoC = ladoC;
}

@Override
public String getNomeFigura() {
return "Triangulo";
}

@Override
public int getArea() {
int area = 0;
area = (base * altura) / 2;
return area;
}

@Override
public int getPerimetro() {
int perimetro = 0;
perimetro = ladoA + ladoB + ladoC;
return perimetro;
}
}
```

Como é possível ver acima, ambas as classes seguiram o contrato da interface *FiguraGeometrica*, porém cada uma delas a implementou de maneira diferente.

Ao contrário da herança que limita uma classe a herdar somente uma classe pai por vez, é possível que uma classe implemente varias interfaces ao mesmo tempo.

Polimorfismo

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).
Se você tem uma *class Animal* sabe que todo animal come, sendo que Cães por exemplo comem ração e Tigres carne. Você pode chamar o método comer nessas 2 classes mesmo sabendo que elas se comportam diferentemente.

```
public  class Animal {

public void comer() {
System.out.println( "Animal Comendo..." );
}
}
public  class Cao extends Animal {
public void comer() {
System.out.println( "Cão Comendo..." );
}
}
public  class Tigre extends Animal {
public void comer() {
System.out.println( "Tirgre Comendo..." );
}
}
```

No caso a sua chamado de método polimórfico ficaria assim:

```
public class  Test {

public void fazerAnimalComer( Animal animal ) {
animal.comer();
}
public static void main( String[] args ) {
Test t = new Test();
t.fazerAnimalComer( new Animal() );
t.fazerAnimalComer( new Cao() );
t.fazerAnimalComer( new Trigre() );
}
}
```

Você vai notar que cada chamada vai fazer uma coisa diferente, porém como existe herança entre as classes todos os métodos vão funcionar, porque são do tipo do mais genérico (Animal) ou são filhos do mais genérico.

Exceptions

O uso de exceções permite separar a detecção da ocorrência de uma situação excepcional do seu tratamento, ao se programar um método em Java.
Alguns possíveis motivos externos para ocorrer uma exceção são:

- Tentar abrir um arquivo que não existe.
- Tentar fazer consulta a um banco de dados que não está disponível.
- Tentar escrever algo em um arquivo sobre o qual não se tem permissão de escrita.
- Tentar conectar em servidor inexistente.

Alguns possíveis erros de lógica para ocorrer uma exceção são:

- Tentar manipular um objeto que está com o valor nulo.
- Dividir um número por zero.
- Tentar manipular um tipo de dado como se fosse outro.
- Tentar utilizar um método ou classe não existentes.

Na linguagem Java existem dois tipos de exceções, que são:

- **Implícitas**: Exceções que não precisam de tratamento e demonstram serem contornáveis. Esse tipo origina-se da subclasse Error ou RunTimeException.
- **Explícitas**: Exceções que precisam ser tratadas e que apresentam condições incontornáveis. Esse tipo origina do modelo throw e necessita ser declarado pelos métodos. É originado da subclasse Exception ou IOException.

Existe também a formação de erros dos tipos throwables que são:

- **Checked Exception:** Erros que acontecem fora do controle do programa, mas que devem ser tratados pelo desenvolvedor para o programa funcionar.
- **Unchecked (Runtime):** Erros que podem ser evitados se forem tratados e analisados pelo desenvolvedor. Caso haja um tratamento para esse tipo de erro, o programa acaba parando em tempo de execução (Runtime).
- **Error:** Usado pela JVM que serve para indicar se existe algum problema de recurso do programa, tornando a execução impossível de continuar.

Para tratar as exceções em Java são utilizados os comandos **try e catch.**
Praticamente, o uso dos blocos try/catch se dá em métodos que envolvem alguma manipulação de dados, bem como:

- CRUD no banco de dados;

- Índices fora do intervalo de array;
- Cálculos matemáticos;
- I/O de dados;
- Erros de rede;
- Anulação de objetos;
- Entre outros;
  Exemplo da utilização do bloco try/catch:

```
public static void main(String args[])
{
String frase = null;
String novaFrase = null;

try
{
novaFrase = frase.toUpperCase();
}
catch (NullPointerException e) //CAPTURA DA POSSÍVEL exceção.
{
//TRATAMENTO DA exceção
System.out.println("O frase inicial está nula,
para solucional tal o problema, foi lhe atribuito um valor default.");
frase = "Frase vazia";
novaFrase = frase.toUpperCase();
}
System.out.println("Frase antiga: "+frase);
System.out.println("Frase nova: "+novaFrase);
}
```

  Quando este código for executado, o mesmo lançará uma NullPointerException, porém esta exceção será tratada desta vez, sendo a mesma capturada pelo catch{} e dentro deste bloco as devidas providências são tomadas. Neste caso é atribuído um valor default à variável frase.

Estrutura try-catch-finally

Como vimos anteriormente, usamos try-catch para tratar uma exceção. A terceira parte dessa estrutura, **finally**, especifica um trecho de código que será sempre executado, não importando o que acontecer dentro do bloco try-catch.
Não é possível deixar um bloco **try-catch-finally** sem executar sua parte **finally**.
Vejamos um exemplo para este bloco:

```
void readFile(String name) throws IOException {
FileReader file = null;
try { file = new FileReader(name); ... // lê o arquivo }
catch (Exception e)
{ System.out.println(e); }
finally {
if (file != null)
file.close(); } }
```

Comandos throw e throws

As cláusulas **throw e throws** podem ser entendidas como ações que propagam exceções, ou seja, em alguns momentos existem exceções que não podem ser tratadas no mesmo método que gerou a exceção. Nesses casos, é necessário propagar a exceção para um nível acima na pilha.

```
public classTesteString {
private static void aumentarLetras() throws NullPointerException //lançando
{
String frase = null;
String novaFrase = null;
novaFrase = frase.toUpperCase();

System.out.println("Frase antiga: "+frase);
System.out.println("Frase nova: "+novaFrase);
}

public static void main(String args[])
{
try
{
aumentarLetras();
}
catch (NullPointerException e)
{
System.out.println("Ocorreu um NullPointerException ao executar o método au
}
```

```
}
}
```

Enquanto isso, a cláusula **throw** cria um novo objeto de exceção que é lançada.

```
public class Exemplo_Throw {

public static void saque( double valor) {
if (valor >400) {
IllegalArgumentException erro = new IllegalArgumentException();

throw` `erro;
} else {
System.out.println("Valor retirado da conta: R$"+valor);
}
}
public static void main(String[] args) {
saque(1500);
}
}
```

Métodos para captura de erros

A classe **Throwable** oferece alguns métodos que podem verificar os erros reproduzidos, quando gerados para dentro das classes. Esse tipo de verificação é visualizado no rastro da pilha (stracktrace), que mostra em qual linha foi gerada a exceção. Abaixo estão descritos os principais métodos que podem ser tratados no bloco catch para visualizar em que momento foi gerado o erro.

- **printStrackTrace** – Imprime uma mensagem da pilha de erro encontrada em um exceção.
- **getStrackTrace** – Recupera informações do stracktrace que podem ser impressas através do método printStrackTrace.
- **getMessage** – Retorna uma mensagem contendo a lista de erros armazenadas em um exceção no formato String.
  No exemplo abaixo, a variável idade, como mostrado, espera um tipo inteiro. Para simular o erro, basta rodar esse código inserindo uma letra/palavra qualquer.

import java.util.InputMismatchException;
import java.util.Scanner;

public class Exemplo_PrintStrackTrace {

```
public static void main(String[] args) {

Scanner sc = new Scanner(System.in);
try {
System.out.print("Digite a idade: ");
int idade = sc.nextInt();
System.out.println(idade);
} catch (InputMismatchException e) {
e.printStackTrace();
}
}
}
```

Collections

A **coleção em Java** é uma estrutura que fornece uma arquitetura para armazenar e manipular o grupo de objetos.

O Java Collections pode realizar todas as operações que você executa em dados, como pesquisa, classificação, inserção, manipulação e exclusão.

Coleção Java significa uma única unidade de objetos. A estrutura Java Collection fornece muitas interfaces (Set, List, Queue, Deque) e classes (ArrayList , Vector, LinkedList , PriorityQueue , HashSet, LinkedHashSet, TreeSet).

Algumas *Collections* são ordenadas, outras organizadas. Na API elas estão organizadas como na seguinte instrução:
Organizada: LinkedHashSet, ArrayList, Vector, LinkedList, LinkedHashMap
Não organizada: HashSet, TreeSet, PriorityQueue, HashMap, HashTable, TreeMap
Ordenada: TreeSet, PriorityQueue, TreeMap
Não ordenada: HashSet, LinkedHashSet, ArrayList, Vector, LinkedList, HashMap, HashTable, LinkedHashMap.

Collections Framework

Collections Framework é um conjunto bem definido de interfaces e classes para representar e tratar grupos de dados como uma única unidade, que pode ser chamada coleção, ou collection. A Collections Framework contém os seguintes elementos:

- **Interfaces**: tipos abstratos que representam as coleções. Permitem que coleções sejam manipuladas tendo como base o conceito “Programar para interfaces e não para implementações”, desde que o acesso aos objetos se restrinja apenas ao uso de métodos definidos nas interfaces;
- **Implementações**: são as implementações concretas das interfaces;
- **Algoritmos**: são os métodos que realizam as operações sobre os objetos das coleções, tais como busca e ordenação.

Vamos ver a hierarquia da estrutura da coleção. O pacote **java.util** contém todas as classes e interfaces para a estrutura Collection.

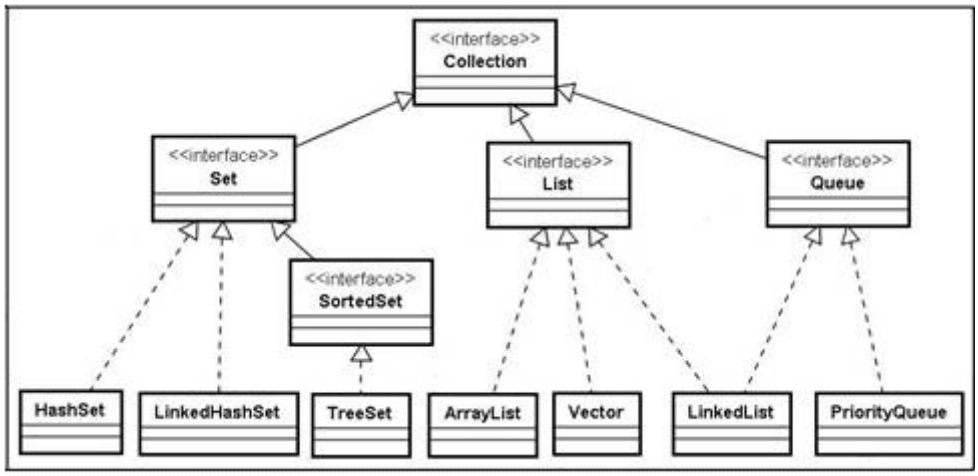


Figura 2. Hierarquia de interfaces e classes

A hierarquia da Collections Framework tem uma segunda árvore. São as classes e interfaces relacionadas a mapas, que não são derivadas de Collection. Essas interfaces, mesmo não sendo consideradas coleções, podem ser manipuladas como tal.

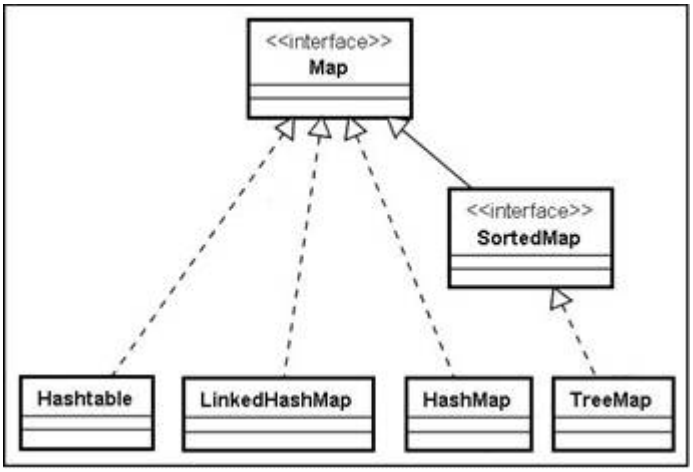


Figura 2. Hierarquia de mapas

Métodos de interface de coleção

Existem muitos métodos declarados na interface Collection. Eles são os seguintes:

• **Collection:** é a raiz das coleções. A plataforma Java não fornece nenhuma

implementação direta dessa interface, normalmente ela é usada quando queremos buscar o máximo de generalização possível, ou seja, não importa se o objeto será do tipo Set ou List.
• **Set:** é uma coleção que não irá aceitar elementos duplicados, conhecida como conjuntos de itens. Por exemplo, clientes de uma loja, carros de uma locadora de veículos, etc. Tudo que você pensar que não podem existir dois ou mais iguais em um mesmo conjunto, são candidatos a pertencerem a um Set.
•**List:** coleções deste tipo aceitam elementos duplicados e são ordenados em uma sequência, normalmente a de inserção. Cada objeto na lista recebe um índice(um inteiro) por posição. Usamos List para criar listas de itens de um pedido de compra, por exemplo.
•**Map:** são itens que possuem uma identificação exclusiva, ou seja, possuem chaves para acessar cada valor do mapa. A chave não pode ser repetida, tem que ser única. Podemos usar Map para criar uma lista telefônica, onde cada chave é o número do telefone e o nome é o valor.
•**SortedSet:** é um Set que mantém a ordem crescente dos elementos dentro dela. São usadas quando precisamos tirar proveito da ordenação, como armazenar os estados brasileiros em ordem alfabética.
•**SortedMap:** um Map que mantém as chaves ordenadas. É útil para mantermos dicionários, por exemplo.
• **Queue:** – um tipo de coleção para manter uma lista de prioridades, onde a ordem dos seus elementos, definida pela implementação de Comparable ou Comparator, determina essa prioridade. Com a interface fila pode-se criar filas e pilhas;

A API oferece também interfaces que permitem percorrer uma coleção derivada de Collection. Neste artigo falaremos de:

- **Iterator** – possibilita percorrer uma coleção e remover seus elementos;
- **ListIterator** – estende Iterator e suporta acesso bidirecional em uma lista, modificando e/ou removendo elementos.

Implementações

A seguir apresentamos algumas características das implementações que podem ajudar a decidir qual delas utilizar em uma aplicação:

- **ArrayList** – é como um array cujo tamanho pode crescer. A busca de um elemento é rápida, mas inserções e exclusões não são. Podemos afirmar que as inserções e exclusões são lineares, o tempo cresce com o aumento do tamanho da estrutura. Esta implementação é preferível quando se deseja acesso mais rápido aos

elementos. Por exemplo, se você quiser criar um catálogo com os livros de sua biblioteca pessoal e cada obra inserida receber um número sequencial (que será usado para acesso) a partir de zero;

- **LinkedList** – implementa uma lista ligada, ou seja, cada nó contém o dado e uma referência para o próximo nó. Ao contrário do ArrayList, a busca é linear e inserções e exclusões são rápidas. Portanto, prefira LinkedList quando a aplicação exigir grande quantidade de inserções e exclusões. Um pequeno sistema para gerenciar suas compras mensais de supermercado pode ser uma boa aplicação, dada a necessidade de constantes inclusões e exclusões de produtos;
- **HashSet** – o acesso aos dados é mais rápido que em um TreeSet, mas nada garante que os dados estarão ordenados. Escolha este conjunto quando a solução exigir elementos únicos e a ordem não for importante. Poderíamos usar esta implementação para criar um catálogo pessoal das canções da nossa discografia;
- **TreeSet** – os dados são classificados, mas o acesso é mais lento que em um HashSet. Se a necessidade for um conjunto com elementos não duplicados e acesso em ordem natural, prefira o TreeSet. É recomendado utilizar esta coleção para as mesmas aplicações de HashSet, com a vantagem dos objetos estarem em ordem natural;
- **LinkedHashSet** – é derivada de HashSet, mas mantém uma lista duplamente ligada através de seus itens. Seus elementos são iterados na ordem em que foram inseridos. Opcionalmente é possível criar um LinkedHashSet que seja percorrido na ordem em que os elementos foram acessados na última iteração. Poderíamos usar esta implementação para registrar a chegada dos corredores de uma maratona;
- **HashMap** – baseada em tabela de espalhamento, permite chaves e valores null. Não existe garantia que os dados ficarão ordenados. Escolha esta implementação se a ordenação não for importante e desejar uma estrutura onde seja necessário um ID (identificador). Um exemplo de aplicação é o catálogo da biblioteca pessoal, onde a chave poderia ser o ISBN (International Standard Book Number);
- **TreeMap** – implementa a interface SortedMap. Há garantia que o mapa estará classificado em ordem ascendente das chaves. Mas existe a opção de especificar uma ordem diferente. Use esta implementação para um mapa ordenado. Aplicação semelhante a HashMap, com a diferença que TreeMap perde no quesito desempenho;
- **LinkedHashMap** – mantém uma lista duplamente ligada através de seus itens. A ordem de iteração é a ordem em que as chaves são inseridas no mapa. Se for necessário um mapa onde os elementos são iterados na ordem em que foram inseridos, use esta implementação. O registro dos corredores de uma maratona, onde a chave seria o número que cada um recebe no ato da inscrição, é um exemplo de aplicação desta coleção.

Interface List

List tem duas implementações – **ArrayList** e **LinkedList**. ArrayList oferece acesso aleatório rápido através do índice. Já em LinkedList o acesso aleatório é lento e necessita de um objeto nó para cada elemento, que é composto pelo dado propriamente dito e uma referência para o próximo nó, ou seja, consome mais memória.

ArrayList

Ele usa uma matriz dinâmica para armazenar o elemento duplicado de diferentes tipos de dados. A classe ArrayList mantém a ordem de inserção e não é sincronizada. Os elementos armazenados na classe ArrayList podem ser acessados aleatoriamente. Considere o seguinte exemplo.

```
import java.util.*;

public class ListaAluno {

  public static void main(String[] args) {
    List<String> lista = new ArrayList<String>();
    lista.add("João da Silva");
    lista.add("Antonio Sousa");
    lista.add("Lúcia Ferreira");
    System.out.println(lista);
  }

}
```

Figura 2. Exemplo de ArrayList

Podemos também classificar em ordem ascendente por exemplo:

```
import java.util.*;

public class ListaAluno {
public static void main(String[] args) {

List<String> lista = new ArrayList<String>();
lista.add("João da Silva");
lista.add("Antonio Sousa");
lista.add("Lúcia Ferreira");
System.out.println(lista);
Collections.sort(lista);
System.out.println(lista);
}
}
```

Utilizamos a classe **Collections** onde possui o método **sort()** que pode classificar a interface List em ordem natural.

Uma outra forma de organizarmos a lista é adicionando um novo requisito. Vamos supor que na classe Aluno além do seu nome, criarmos o nome do curso e a sua respectiva nota, para isso redefinimos a classe *Aluno* e iremos modificar a classe *ListaAluno*, adicionando novos objetos Aluno e não só string.

```
public class Aluno {

private String nome;
private String curso;
double nota;

Aluno(String nome, String curso, double nota) {
this.nome = nome;
this.curso = curso;
this.nota = nota;
}

public String toString() {
return this.nome;
}
// Métodos getters e setters
}

import java.util.*;
public class ListaAluno {

public static void main(String[] args) {

List<Aluno> lista = new ArrayList<Aluno>();

Aluno a = new Aluno("Carlos Ferreira", "JavaJoão da Silva", "Linux básico"
Aluno b = new Aluno( "MariaAntonio Souzsa", "PythonOpenOffice", 0);
Aluno c = new Aluno( "Carla Silva", "CloudLúcia Ferreira", "Internet", 0);

lista.add(a);
lista.add(b);
lista.add(c);
System.out.println(lista);
}
}
```

Uma outra situação que podemos encontrar é a comparação de objetos, isso é feito com o **compareTo( )** .
Na classe *Aluno* consideraremos que a comparação entre dois objetos será determinada pela comparação entre seus nomes, que são do tipo String.

```
public class Aluno implements Comparable<Aluno>{
private String nome;
private String curso;
double nota;

Aluno(String nome, String curso, double nota) {
this.nome = nome;
this.curso = curso;
this.nota = nota;
}
public String toString() {
return this.nome;
}
public int compareTo(Aluno aluno) {
return this.nome.compareTo(aluno.getNome());
}
// Métodos getters e setters
public String getNome() {
return this.nome;
}
}
```

Note que no método **compareTo()** fizemos simplesmente uma chamada ao mesmo método, só que para o atributo nome, que é do tipo String. String é uma classe comparável, isto é, já implementa *Comparable*.

Agora podemos incluir uma chamada ao método **sort()** na classe *ListaAluno*. A ordenação implementada por *Comparable* é chamada ordenação natural. Por exemplo, em uma String a ordenação natural é a ordem alfabética, em uma classe **wrapper** – **Integer**, **Float**, etc. – a ordenação natural é a ordem numérica.

Mas em algumas situações a classe de objetos pode não ser comparável, ou seja, não podemos implementar o *Comparable*, nessas situações teremos que utilizar a interface **Comparator**. Nessa interface tem um método chamado **compare( )** , ele recebe dois objetos que são comparados e retorna um inteiro negativo, zero ou um inteiro positivo se o primeiro objeto é menor, igual ou maior que o segundo. Veja a implementação:

```
import java.util.Comparator;
public class ComparaAluno implements Comparator<Aluno> {
```

```
public int compare(Aluno a, Aluno b) {
return a.getNome().compareTo(b.getNome());
}
}
```

Para usar esta implementação chamamos o método sobrecarregado *sort()* da classe *Collections*. Ele recebe como argumentos a lista a ser ordenada e uma instância da implementação de *Comparator*.

```
import java.util.*;

public class ListaAluno {

public static void main(String[] args) {

List<Aluno> lista = new ArrayList<Aluno>();

ComparaAluno ca = new ComparaAluno();

Aluno a = new Aluno("Carlos Ferreira", "Java básico" , 0);
Aluno b = new Aluno( "Maria Souza", "Python", 0);
Aluno c = new Aluno( "Carla Silva", "Cloud", 0);

lista.add(a);
lista.add(b);
lista.add(c);
System.out.println(lista);
Collections.sort(lista, ca);
System.out.println(lista);
}
}
```

Uma outra interface que utilizamos é a *Interator*. As interfaces que estendem Collection herdam o método *iterator()*. Quando este método é chamado por um collection ele retorna uma interface *Iterator*. Após essa chamada, usamos os métodos de *Iterator* para percorrer um *collection* do início ao fim e até remover seus elementos. Veja o exemplo abaixo:

```
import java.util.*;

public class ListaAluno {
public static void main(String[] args) {

List<Aluno> lista = new ArrayList<Aluno>();
```

```
Aluno a = new Aluno("Carlos Ferreira", "Java básico" , 0);
Aluno b = new Aluno( "Maria Souza", "Python", 0);
Aluno c = new Aluno( "Carla Silva", "Cloud", 0);
Aluno d = new Aluno( "José Guerreiro", "MySQL", 0);
lista.add(a);
lista.add(b);
lista.add(c);
lista.add(d);
System.out.println(lista);
Aluno aluno;
Iterator<Aluno> itr = lista.iterator();
while (itr.hasNext()) {
aluno = itr.next();
System.out.println(aluno.getNome());
}
}
}
```

Observe que é necessário informar o tipo que será retornado pelo *Iterator*. O método *hasNext()* retorna true se houver elemento a ser lido, e o método *next()* retorna o objeto, de acordo com o tipo informado na declaração da interface.

Uma outra interface que iremos utilizar é o **set( )** diferente do *List ( )* ela não permite elementos duplicados. Para isso no exemplo abaixo vamos forçar a inclusão de um elemento duplicado.

```
import java.util.*;
public class ListaAluno {

public static void main(String[] args) {

Set<Aluno> conjunto = new HashSet<Aluno>();
Aluno a = new Aluno("Carlos Ferreira", "Java básico" , 0);
Aluno b = new Aluno( "Maria Souza", "Python", 0);
Aluno c = new Aluno( "Carla Silva", "Cloud", 0);
Aluno d = new Aluno( "Maria Souza", "Python", 0);

conjunto.add(a);
conjunto.add(b);
conjunto.add(c);
conjunto.add(d);

System.out.println(conjunto);
}
}
```

HashSet usa o código hash do objeto – dado pelo método hashCode() – para saber onde deve por e onde buscar o mesmo no conjunto (Set). Antes ele verifica se não existe outro objeto no Set com o mesmo código hash. Se não há código hash igual, então ele sabe que o objeto a ser inserido não está duplicado. Dessa forma, classes cujas instâncias são elementos de HashSet devem implementar o método hashCode(). Como consequência disso, a classe Aluno, no nosso exemplo, deve sobrescrever o método hashCode().

Conforme o contrato geral de hashCode(), que consta na especificação da classe Object, se dois objetos são diferentes de acordo com equals() então não é obrigatório que seus códigos hash sejam diferentes.

Portanto, objetos que retornam o mesmo código hash não são necessariamente iguais. Assim, quando encontra no conjunto um objeto com o mesmo código hash do objeto a ser inserido, HashSet faz uma chamada ao método equals() para verificar se os dois objetos são iguais. Dessa forma, a classe Aluno deve sobrescrever o método equals() também.

```
public class Aluno implements Comparable<Aluno>{

private String nome;
private String curso;
double nota;

Aluno(String nome, String curso, double nota) {

this.nome = nome;
this.curso = curso;
this.nota = nota;
}

public String toString() {
return this.nome;
}

public int compareTo(Aluno aluno) {
return this.nome.compareTo(aluno.getNome());
}

public boolean equals(Object o) {
Aluno a = (Aluno) o;
return this.nome.equals(a.getNome());
}
```

```
public int hashCode() {
return this.nome.hashCode();
}
// Métodos getters e setters

public String getNome() {
return this.nome;
}
}
```

E a última interface que vamos mostrar é a **Map** que não estende Collection. Isso causa uma mudança profunda na aplicação, visto que os métodos usados anteriormente não poderão ser usados. Map tem seus próprios métodos para inserir/buscar/remover elementos na estrutura.
Para usar uma classe que implementa Map, quaisquer classes que forem utilizadas como chave devem sobrescrever os métodos *hashCode()* e *equals()*. Isso é necessário porque em um *Map* as chaves não podem ser duplicadas, apesar dos valores poderem ser. Para mostrar a sua prática, utilizamos um **TreeMap**, que garante que as chaves estarão em ordem ascendente, pois agora queremos uma estrutura onde possamos recuperar os dados de um aluno passando apenas o seu nome como argumento de um método. Ou seja, informamos o nome do aluno e o objeto correspondente a esse nome é devolvido.

```
import java.util.*;

public class MapaAluno {

public static void main(String[] args) {

Map<String, Aluno> mapa = new TreeMap<String, Aluno>();

Aluno a = new Aluno("Carlos Ferreira", "Java básico" , 0);
Aluno b = new Aluno( "Maria Souza", "Python", 0);
Aluno c = new Aluno( "Carla Silva", "Cloud", 0);
Aluno d = new Aluno( "José Guerreiro", "MySQL", 0);

mapa.put( "Carlos Ferreira", a);
mapa.put( "Maria Souza", b);
mapa.put( "Carla Silva", c);
mapa.put( "José Guerreiro", d);

System.out.println(mapa);
System.out.println(mapa.get(" José Guerreiro "));

Collection<Aluno> alunos = mapa.values();
```

```
for (Aluno e : alunos) {
System.out.println(e);
}
}
}
```

Note que na declaração do *collection* informamos dois tipos: String e Aluno. O primeiro refere-se à chave e o segundo ao valor. O método para inserir na estrutura é *put()*, que recebe dois objetos (chave e valor). Para recuperar um objeto específico utilizamos o método *get()* passando a chave como parâmetro.

Como *Map* não estende *Collection*, não tem os métodos *iterator()* e *listIterator()*. Entretanto, existe o método *keySet()* que retorna um *Set* com as chaves do mapa, e o método *values()* que retorna um *Collection* com os valores associados às chaves. Assim, podemos percorrer o mapa partindo desses métodos e usando **enhanced-for**. A aplicação deste comando (*for (Objeto obj: colecao) { ... }*) para percorrer o mapa.

Foreach

A utilização é bem simples e direta. Vamos usar essa variante do for que percorre sempre, do começo ao fim, todos os elementos de um Array.
É bem útil, também, em termos de precaução e organização, pois alguns programadores não gostam de usar o índice 0, usam direto o índice 1 do array, ou as vezes nos confundimos e passamos (durante as iterações) do limite do array. Isso pode ser facilmente evitado usando o laço for modificado para percorrer os elementos de um array.
Poderíamos pegar como exemplo, ter uma lista populada e querer percorrer essa lista, ficaria assim:

```
import java.util.Arrays;
import java.util.List;
public  class Sample {
public static void main(String[] args) {
List< Integer > itens = Arrays.asList( 11, 10, 16, 5, 85 ); itens.forEach(i
}
}
```

Dessa maneira já podemos ver que esta bem mais fácil, mas ainda podemos melhorar este código usando o recurso também adicionado ao Java 8, method reference:

```
import java.util.Arrays;
import java.util.List;
public  class Sample {
public static void main(String[] args) {
List<Integer> itens = Arrays.asList( 11, 10, 16, 5, 85 ); itens.forEach(Sys
}
}
```

Ainda podemos filtar itens de List usando Java 8 e lambda, pois ficaria mais fácil de ler e entender:

```
List< Integer > itens = Arrays.asList( 11, 10, 16, 5, 85 );
itens.stream().filter(i -> i == 16).forEach(System.out::println);
```

Um outro exemplo seria filtrar os item acima de 20, podemos escrever assim:

```
List< Integer > itens = Arrays.asList( 11, 10, 16, 5, 85 );

itens.stream().filter(i -> i > 16).forEach(System.out::println);
```

Estes são apenas alguns exemplos, mas ainda é possível fazer muito mais.

Lambda expression

A grande vantagem de funções lambda é diminuir a quantidade de código necessária para a escrita de algumas funções, como por exemplo, as classes internas que são necessárias em diversas partes da linguagem Java, como Listeners e Threads.
Simplificando um pouco a definição, uma função lambda é uma função sem declaração, isto é, não é necessário colocar um nome, um tipo de retorno e o modificador de acesso. A ideia é que o método seja declarado no mesmo lugar em que será usado. As funções lambda em Java tem a sintaxe definida como (argumento) -> (corpo).
Para exemplificar a utilização de expressões lambda com threads será analisado um programa que cria uma thread com uma função interna e que vai apenas mostrar a mensagem “Thread com classe interna!”.

```
Runnable r = () -> System.out.println("Thread com função lambda!");
new Thread(r).start();
```

Essa expressão não passa nenhum parâmetro, pois ela será passada para a função run, definida na interface Runnable, que não tem nenhum parâmetro, então ela também não tem nenhum retorno.

As funções lambdas podem ser bastante utilizadas com as classes de coleções do Java, pois nessas fazemos diversos tipos de funções que consistem basicamente em percorrer a coleção e fazer uma determinada ação, como por exemplo, imprimir todos os elementos da coleção, filtrar elementos da lista e buscar um determinado valor na lista. Um exemplo do que pode ser feito com funções lambda, são expressões matemáticas. Neste exemplo é exibido o código para mostrar o quadrado de todos os elementos de uma lista de números inteiros.

```
System.out.println("Imprime o quadrado de todos os elementos da lista!");

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);

list.forEach(n -> System.out.println(n * n));
```

Uma outra funcionalidade é implementar as funções Lambda com Listeners.

```
button.addActionListener( (e) -> {

System.out.println("O botão foi pressionado, e o código executado utiliza u
//Realiza alguma ação quando o botão for pressionado
});
```

Praticamente qualquer Listener pode ser escrito utilizando expressões lambda, como os para escrever arquivos em logs e para verificar se um atributo foi adicionado em uma sessão de um usuário em aplicação web.

Optional

Nota da API:

Optional destina-se principalmente ao uso como um tipo de retorno de método, onde há uma clara necessidade de representar “nenhum resultado” e onde o uso `null` provavelmente causa erros. Uma variável cujo tipo é `Optional` nunca deve ser ela mesma `null`; deve sempre apontar para uma `Optional` instância.
Com isso é possível:

- Evitar erros `NullPointerException`.
- Parar de fazer verificações de valor nulo do tipo `if (cliente != null)`.
- Escrever código mais limpo e elegante.

Primeiro vou apresentar alguns métodos e em seguida alguns cenários de uso.

empty - Retorna uma instância de `Optional` vazia.

> Optional < **Cliente** > retorno = Optional.**empty**();

of - Retorna um `Optional` com o valor fornecido, mas o valor não pode ser nulo. Se não tiver certeza de que o valor não é nulo use `Optional.ofNullable`.

> Optional<**Cliente**> retorno = Optional.**of**(buscaCliente(cpf));

ofNullable - Se um valor estiver presente, retorna um `Optional` com o valor , caso contrário, retorna um `Optional` vazio. Este é um dos métodos mais indicados para criar um `Optional`.

Optional<**Cliente**> retorno = Optional.**ofNullable**(buscaCliente(cpf));

filter - Se um valor estiver presente e o valor corresponder ao predicado retorna um `Optional` com o valor, se não, retorna um `Optional` vazio.

Optional< **Cliente** > retorno = buscaCliente(cpf)
.**filter**(cliente -> !cliente.getNome().isEmpty());

isPresent - Se um valor estiver presente retorna `true`, se não, retorna `false`.

```
Optional< Cliente > retorno = Optional.ofNullable(buscaCliente(cpf));
if (retorno.isPresent()) {
   System.out.println("Cliente encontrado.");
} else {
   System.out.println("Cliente não encontrado.");
}
```

O código acima não é o cenário ideal do uso de `Optional`.

get - Se um valor estiver presente retorna o valor, caso contrário, lança `NoSuchElementException`. Então para usar `get` é preciso ter certeza de que o `Optional` não está vazio.

```
Optional< Cliente > retorno = Optional.ofNullable(buscaCliente(cpf));
if (retorno.isPresent()) {
   Cliente cliente = retorno.get();
}
```

ifPresent - Se um valor estiver presente, executa a ação no valor, caso contrário, não faz nada.

```
public void login(String cpf, String senha) {
   dao.buscaPorCPF(cpf).
      ifPresent (cliente -> cliente.realizaLogin(senha));
}
```

map - Se um valor estiver presente retorna um `Optinal` com o resultado da aplicação da função de mapeamento no valor, caso contrário, retorna um `Optional` vazio.

```
if (optCliente.isPresent()) {
   String nome = optCliente.map(Cliente::getNome);
}
```

flatMap - Semelhante a `map`, se um valor estiver presente, retorna o resultado da aplicação da função de mapeamento no valor, caso contrário retorna um `Optional` vazio. A diferença é que `flatMap` pode se aplicado a um mapeamento que também retorna um `Optional`.

```
Endereco endereco =
buscaCliente(cpf).flatMap(Cliente::getEndereco).get();
```

Algumas outras aplicações `Optional` para tratar o cenário onde um cliente não é encontrado de várias maneiras interessantes.

orElse - Se um valor estiver presente, retorna o valor, caso contrário, retorna o valor definido no parâmetro.

```
//Método na classe ClienteService alterado
public String getNomePorCPF(String cpf) {
   return dao.buscaPorCPF(cpf).map(Cliente::getNome)
      .orElse("DESCONHECIDO");
}
```

orElseGet - Se um valor estiver presente, retorna o valor, caso contrário, retorna o resultado produzido pelo parâmetro.

```
//Método na classe ClienteService alterado
public String getNomePorCPF(String cpf) {
   return dao.buscaPorCPF(cpf).map(Cliente::getNome)
      .orElseGet(Constantes::getNomePadrao);
}
```

orElseThrow - Se um valor estiver presente, retorna o valor, caso contrário, lança a exceção informada no parâmetro.

```
//Método na classe ClienteService alterado
public String getNomePorCPF(String cpf) {
   return dao.buscaPorCPF(cpf).map(Cliente::getNome)
      .orElseThrow(IllegalArgumentException::new);
}
```

ifPresentOrElse - Se um valor estiver presente, executa uma determinada ação, caso contrário, executa uma outra ação.

Veja o código abaixo:

```
//Método na classe ClienteService
public void login(String cpf, String senha) {
   Cliente cliente = dao.buscaPorCPF(cpf);
   if (cliente != null) {
      cliente.realizaLogin(senha);
   } else {
      registraFalhaLogin();
   }
}
```

Com `ifPresentOrElse` o código pode ser escrito assim:

```
//Método na classe ClienteService alterado
public void login(String cpf, String senha) {
   dao.buscaPorCPF(cpf).ifPresentOrElse(cliente ->
      cliente.realizaLogin(senha), () -> registraFalhaLogin());
}
```

or - Se um valor estiver presente, retorna o valor, caso contrário, executa uma ação. Por exemplo, é possível tentar obter um cliente do cache e no caso do retorno estar vazio tentar obter o cliente do banco de dados.

```
Optional<Cliente> cliente = cache.getCliente(cpf).or(() -> dao.buscaPorCPF(
```

stream - Se um valor estiver presente, retorna uma `Stream` contendo apenas esse valor, caso contrário, retorna uma `Stream` vazia.

```
dao.buscaPorCPF(cpf).stream();
```

DateTime operators

A partir do Java 8 foi criado uma nova API para facilitar o uso da data e tempo do sistema. Esta nova API aparecerá no novo pacote Java SE 8 `java.time`.
A nova API é orientada por três ideias principais:

- **Classes de valor imutável.** Uma das sérias fraquezas dos formatadores existentes em Java é que eles não são seguros para threads. Isso sobrecarrega os desenvolvedores de usá-los de maneira segura para threads e pensar em problemas de simultaneidade no desenvolvimento diário do código de manipulação de datas. A nova API evita esse problema, garantindo que todas as suas classes principais sejam imutáveis e representem valores bem definidos.

- **Design orientado a domínio.** A nova API modela seu domínio com muita precisão com classes que representam diferentes casos de uso para `Date` e de `Time` perto. Isso difere das bibliotecas Java anteriores que eram bastante pobres nesse sentido. Por exemplo, `java.util.Date` representa um instante na linha do tempo - um invólucro em torno do número de milissegundos desde a época do UNIX - mas se você ligar `toString()`, o resultado sugere que ele possui um fuso horário, causando confusão entre os desenvolvedores.

  Essa ênfase no design orientado a domínio oferece benefícios a longo prazo em termos de clareza e compreensibilidade, mas talvez seja necessário pensar no modelo de datas do domínio do aplicativo ao migrar de APIs anteriores para o Java SE 8.

- **Separação de cronologias.** A nova API permite que as pessoas trabalhem com diferentes sistemas de calendário para atender às necessidades dos usuários em algumas áreas do mundo, como Japão ou Tailândia, que não seguem necessariamente a ISO-8601. Isso é feito sem impor encargos adicionais à maioria dos desenvolvedores, que precisam trabalhar apenas com a cronologia padrão.

LocalDate e LocalTime

Eles são locais no sentido de que representam data e hora do contexto do observador, como um calendário em uma mesa ou um relógio na parede. Há também uma classe composta chamada `LocalDateTime`, que é um emparelhamento de `LocalDate` e `LocalTime`.
Segue as principais classes:

Class	Description
`LocalDate`	Represents a date (year, month, day (yyyy-MM-dd))
`LocalTime`	Represents a time (hour, minute, second and milliseconds (HH-mm-ss-zzz))
`LocalDateTime`	Represents both a date and a time (yyyy-MM-dd-HH-mm-ss.zzz)
`DateTimeFormatter`	Formatter for displaying and parsing date-time objects

Figura 3029. Classes de Data e Hora

Vamos começar a inserir essas classes. Por exemplo, para mostrar a data atual e a hora atual e os dois ao mesmo tempo:

```
import java.time.*;
public class MinhaClasse {
  public static void main(String[] args) {
    LocalDate myObj = LocalDate.now();
    LocalTime myObj1 = LocalTime.now();
    LocalDateTime myObj2 = LocalDateTime.now();
    System.out.println(myObj);
    System.out.println(myObj1);
    System.out.println(myObj2);
  }
}
```

Quando você executa por exemplo a classe *LocalDateTime.now()* pode surgir um resultado como esse:

2020-04-15T17: 26: 04.120902

O “T” no exemplo acima é usado para separar a data da hora. Você pode usar a `DateTimeFormatter` classe com o `ofPattern()` método no mesmo pacote para formatar ou analisar objetos de data e hora. O exemplo a seguir removerá o “T” e os milissegundos da data e hora:

```
import java.time.LocalDateTime; // Import the LocalDateTime class
import java.time.format.DateTimeFormatter; // Import the DateTimeFormatter

public class MyClass {
  public static void main(String[] args) {
    LocalDateTime myDateObj = LocalDateTime.now();
    System.out.println("Antes da formatação: " + myDateObj);
    DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-yyyy

    String formattedDate = myDateObj.format(myFormatObj);
    System.out.println("Depois da formatação: " + formattedDate);
```

```
    }
}
```

O resultado é mostrado a seguir:

Antes da formatação: 2020-04-15T17: 28: 46.782388
Após a formatação: 15-04-2020 17:28:46

O `ofPattern()` método aceita todos os tipos de valores, se você deseja exibir a data e a hora em um formato diferente. Veja a tabela abaixo:

Value	Example
yyyy-MM-dd	"1988-09-29"
dd/MM/yyyy	"29/09/1988"
dd-MMM-yyyy	"29-Sep-1988"
E, MMM dd yyyy	"Thu, Sep 29 1988"

Figura 310. Formatação de data e hora