

Estrutura condicional

I. Estrutura condicional simples

```
If ("condição") {  
    Comando  
}
```

O comando só será executado se a condição for verdadeira. Uma condição é uma comparação que possui dois valores possíveis: verdadeira ou falsa.

```
If ("condição") {  
    Comando1  
    Comando2  
    Comando3  
}
```

Os comandos 1, 2 e 3 só serão executados se a condição for verdadeira. As “{” e “}” indicam início e o fim do comando if()

II. Estrutura condicional composta

```
If ("condição") {  
    Comando1  
} Else {  
    Comando2  
}
```

Se a condição for verdadeira, será executado o “comando1”; caso contrário, será executado o “comando2”.

III. Estrutura case

Em alguns programas, existem situações mutuamente exclusivas, isto é, se uma situação for executada, as demais não serão. Quando for este o caso, um comando seletivo será o mais indicado, e esse comando, em “C”, tem a seguinte sintaxe:

Se o seletor atingir a lista de alvo1, o comando1 será executado; se atingir a lista de alvo2, o comando2 será executado. Se nenhum alvo for atingido, nada será executado.

O comando switch (variável) avalia o valor de uma variável para decidir qual case será executada. Cada case está associado a UM possível valor da variável, que deve ser, obrigatoriamente, do **tipo char, unsigned char, int, unsigned int, short int, long ou unsigned long**.

O comando break deve ser utilizado para impedir a execução dos comandos definidos nos cases subsequentes.

```
Switch ("seletor") {  
    Case "alvo1"  
        Comando1;  
        Break;  
    Case "alvo2"  
        Comando2;  
        Break;  
    Default:  
        Break;  
}
```

IV. Operadores lógicos

Os principais operadores lógicos são: &&, || e !, que significam respectivamente e, ou, não e são usados para conjunção, disjunção e negação.

TABELA E	TABELA OU	TABELA NÃO
V e V = V	V e V = V	!V = F
V e F = F	V e F = V	!F = V
F e V = F	F e V = V	
F e F = F	F e F = F	

Na linguagem Java, todas as condições devem estar entre parênteses.

Exemplos:

```
If ( x == 3 ) {  
    System.out.println("Número igual a 3");  
}
```

No exemplo acima, existe apenas uma condição que, obrigatoriamente, deve estar entre parênteses.

```
If ( x > 5 && x < 10 ) {  
    System.out.println("Número entre 5 e 10");  
}
```

No exemplo acima, existe mais de uma condição, as quais, obrigatoriamente, devem estar entre parênteses.

```

If ( (x == 5 && y == 2) || y == 3 ) {

    System.out.println("x é igual a 5 e y é igual a 2, ou y é igual a 3");

}

```

No exemplo acima, existem mais de uma condição e mais de um tipo de operador lógico, logo, além dos parênteses que envolvem todas as condições, devem existir ainda parentes que indiquem a prioridade de execução das condições. Nesse exemplo, as condições com o operador &&, ou seja, (x == 5 && y == 2), serão testadas, e seu resultado será testado com a condição || y == 3.

```

If ( x == 5 && (y ==2 || y == 3) ) {

    System.out.println("x é igual a 5, e y é igual a 2 ou y é igual a 3");

}

```

No exemplo acima, existe mais de uma condição e mais de um tipo de operador lógico, logo, além dos parênteses que envolvem todas as condições, devem existir ainda parênteses que indiquem a prioridade de execução das condições. Nesse exemplo, as condições com o operador ||, ou seja, (y == 2 || y == 3), serão testadas, e seu resultado será testado com a condição && x == 5.

Exercício exemplo:

A nota final de um estudante é calculada a partir de três notas atribuídas, respectivamente, a um trabalho de laboratório, a uma avaliação semestral e a um exame final. A média das três notas mencionadas obedece aos pesos a seguir:

NOTA	PESO
Trabalho de laboratório	2
Avaliação semestral	3
Exame final	5

Faça um programa que receba as três notas, calcule e mostre a média ponderada e o conceito que segue a tabela.

MÉDIA PONDERADA	CONCEITO
8.0 e 10.0	A
7.0 e 8.0	B
6.0 e 7.0	C
5.0 e 6.0	D
0.0 e 5.0	E

```
1  /*
2   * Exemplo exercicio
3   */
4
5  package com.mycompany.exemploexercicio;
6
7  /**
8   *
9   * @author clsma
10  */
11 public class ExemploExercicio {
12
13     public static void main(String[] args) {
14         double nota1, nota2, nota3, media;
15
16         // atribuindo valores as variavies
17         nota1 = 7.5;
18         nota2 = 6.7;
19         nota3 = 9.0;
20
21         // executando a media ponderada
22         media = (nota1 * 2 + nota2 * 3 + nota3 * 5) / 10;
23
24         // imprime o resultado
25         System.out.println("Média ponderada: " + media);
26     }
27 }
```

Vetores

Definição de vetor:

Vetor também é conhecido como variável composta **homogênea unidimensional**. Isto quer dizer que se trata de um conjunto de **variáveis de mesmo tipo**, que possuem o mesmo identificador (nome) e são alocadas sequencialmente na memória. Como as variáveis têm o mesmo nome, o que as distingue é um índice que referencia sua localização dentro da estrutura.

Declaração de vetor em Java:

Os vetores em Java são definidos pela existência de colchetes vazios antes ou depois do nome da variável no momento da declaração. Logo depois, deve ser feito o dimensionamento do vetor.

```
tipo_da_variavel [] nome_variavel = new tipo_da_variavel[dimensionamento];
```

Exemplo de vetor:

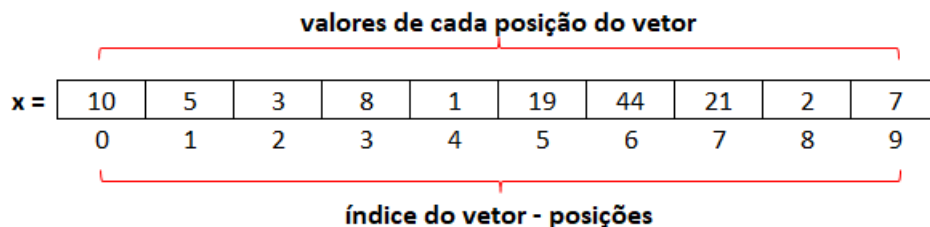
Nos exemplos a seguir são utilizadas duas linhas de comando:

- a primeira declara um vetor e,
- a segunda define o seu tamanho.

Exemplo 1: Na primeira linha deste exemplo, os colchetes vazios após o nome definem que **x** será um vetor. O tipo **int** determina que todas as suas posições armazenarão valores inteiros. A segunda linha determina que o vetor **x** **terá tamanho 10, ou seja, das posições de 0 a 9**.

```
int x[];
```

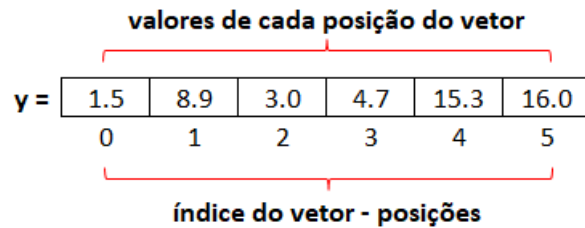
```
x = new int[10];
```



Exemplo 2: Na primeira linha deste exemplo, os colchetes vazios antes do nome definem que **y** será um vetor. O tipo **float** determina o tipo do número que poderá ser armazenado em todas as suas posições. A segunda linha determina que o vetor **y** **terá tamanho 6, ou seja, das posições de 0 a 5**.

```
float [] y;
```

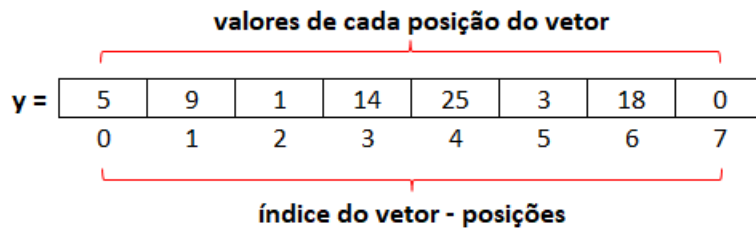
```
y = new float[6];
```



Já nos exemplos apresentados a seguir, utilizou-se a forma condensada, onde a declaração e o dimensionamento de um vetor são feitos utilizando-se uma única linha.

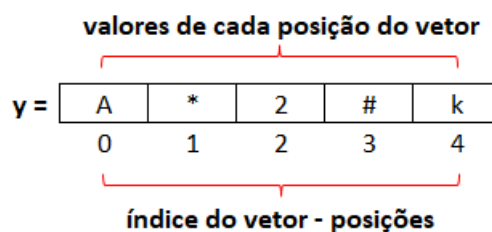
Exemplo 3: Este exemplo define e dimensiona o vetor **y** utilizando uma única linha. Assim, a parte que antecede o sinal de igual informa que **y** é um vetor e que poderá armazenar números inteiros. A parte que sucede o sinal de igual dimensiona o tamanho de **y** em 8 das posições de 0 a 7.

```
int y[] = new int[8];
```



Exemplo 4: Este exemplo define e dimensiona o vetor **y** utilizando uma única linha. Assim, a parte que antecede o sinal de igual informa que **y** é um vetor e que poderá armazenar qualquer caractere. A parte que sucede o sinal de igual dimensiona o tamanho de **y** em 5 das posições de 0 a 4.

```
char [] y new char[5];
```



Atribuindo valores ao vetor:

As atribuições em vetor exigem que seja informada em qual de suas posições o valor ficará armazenado. Deve-se lembrar sempre que a primeira posição de um vetor em Java tem índice 0.

`vet[0] = 1;` → atribui o valor 1 à primeira posição do vetor (lembre-se que o vetor começa na posição 0).

`x[3] = 'b';` → atribui a letra b à quarta posição do vetor (lembre-se que o vetor começa na posição 0).

Preenchendo um vetor:

Preencher um vetor significa atribuir valores a todas as suas posições. Assim, deve-se implementar um mecanismo que controle o valor de índice.

```
Scanner scanner = new Scanner(System.in);
int [] vet = new int[10];
int i;

for (i = 0; i < 10; i++) {
    vet[i] = scanner.nextInt();
}
```

Nesse exemplo, a estrutura de repetição **for** foi utilizada para garantir que a variável **i** assumia todos os valores possíveis para o índice do vetor de 0 a 9. Assim, para cada execução da repetição, uma posição diferente do vetor será utilizada.

Mostrando os elementos do vetor:

Mostrar os valores contidos em um vetor também implica a utilização do índice.

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

Nesse exemplo, a estrutura de repetição **for** foi utilizada para garantir que a variável **i** assumia todos os valores possíveis para o índice do vetor de 0 a 9. Assim, para cada execução da repetição, será utilizada uma posição diferente e, dessa forma, todos os valores do vetor serão mostrados.

```

/*
    Faça um programa que preencha um vetor com nove números, calcule e mostre
    os números ímpares e suas respectivas posições.
*/

```

```

package com.mycompany.vetorexemplo1;

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

public class VetorExemplo1 {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int [] vet = new int[9];
        int i;

        // atribuindo os valores ao vetor
        for(i = 0; i < 9; i++) {

            try {

                System.out.print("Digite o valor do vetor: ");
                vet[i] = sc.nextInt();

            } catch(InputMismatchException e) {
                System.out.println("Erro de digitação!");
                break;
            }

        }

        // apresentar os números ímpares do vetor e suas posições
        for (i = 0; i < 9; i++) {

            if (vet[i] % 2 != 0) {
                System.out.println("Posição: " + i + " Valor: " + vet[i]);
            }

        }

    }

}

```

Saída:

```

Digite o valor do vetor: 3
Digite o valor do vetor: 6
Digite o valor do vetor: 9
Digite o valor do vetor: 8
Digite o valor do vetor: 4
Digite o valor do vetor: 5
Digite o valor do vetor: 2
Digite o valor do vetor: 0
Digite o valor do vetor: 7
Posição: 0 Valor: 3
Posição: 2 Valor: 9
Posição: 5 Valor: 5
Posição: 8 Valor: 7

```


Matriz

Definição de matriz:

Uma matriz pode ser definida como um conjunto de variáveis de mesmo tipo e identificadas pelo mesmo nome. Essas variáveis são diferenciadas por meio da especificação de suas posições dentro dessa estrutura.

A linguagem Java permite a declaração de matrizes unidimensionais (mais conhecidas como vetores – descritos anteriormente), **bidimensionais e multidimensionais**. As matrizes mais utilizadas possuem duas dimensões. Para cada dimensão deve ser adotado um índice.

Declaração de matriz:

Matrizes em Java são definidas pela existência de colchetes vazios antes ou depois do nome da variável no momento da declaração. Logo depois, deve ser feita a definição do tamanho de cada dimensão da matriz.

Para utilizar uma matriz em Java, é necessário seguir dois passos:

1º Passo: DECLARAR UMA VARIÁVEL QUE FARÁ REFERÊNCIA AOS ELEMENTOS.

tipo_de_dados nome_variável [][] ... []; → Os colchetes vazios após o nome da variável definem que a variável será uma estrutura multidimensional.

2º Passo: DEFINIR O TAMANHO DAS DIMENSÕES DA MATRIZ

nome_variavel = new tipo_de_dados[dimensão1][dimensão2][dimensão3] ... [dimensão];

onde:

- **tipo_de_dados** é o tipo de dados que poderá ser armazenado na sequência das variáveis que formam a matriz;
- **nome_variavel** é o nome da variável do tipo matriz;
- **[dimensão1]** representa o tamanho da primeira dimensão da matriz;
- **[dimensão2]** representa o tamanho da segunda dimensão da matriz;
- **[dimensão3]** representa o tamanho da n-ésima dimensão da matriz.

Exemplo de matriz:

Da mesma maneira como ocorre com os vetores, os índices de uma matriz começam sempre em 0 (zero) e podem variar até o tamanho da dimensão menos uma unidade.

É importante ressaltar que, em Java, os pares de colchetes podem aparecer todos antes do nome da variável ou todos depois do nome da variável ou, ainda, alguns antes e outros depois. Assim, todos os exemplos a seguir são válidos.

Exemplo 1:

```
float x[][];
```

```
x = new float[2][6];
```

A declaração anterior criou uma variável chamada **x** contendo duas linhas de 0 a 1 com seis colunas cada de 0 a 5, capazes de armazenar números reais, como pode ser observado a seguir:

variável x

índice linhas da matriz	{	0	1.5	3	0.78	0.3	1.2	9
		1	1	0.9	4.1	4	2.5	32.2
			0	1	2	3	4	5

índice ou posições da matriz

Exemplo 2:

```
char [][] mat;
```

```
mat = new char[4][3];
```

A declaração anterior criou uma variável chamada **mat** contendo quatro linhas de 0 a 3 com três colunas cada de 0 a 2, capazes de armazenar símbolos, como pode ser observado a seguir.

índice

linhas da

matriz

<div style="display: flex; align-items: center;"> <div style="font-size: 3em; margin-right: 5px;">{</div> <div> <p>0</p> <p>1</p> <p>2</p> <p>3</p> </div> </div>	A	3	1	0.3	1.2	9
	*	k	l	v	j	3
	&	s	n	4	t	4
	\$	u)	([]
	0	1	2	3	4	5

índice ou posições da matriz

variável x

Atribuindo valores a uma matriz:

Atribuir valor a uma matriz significa armazenar uma informação em um de seus elementos, identificando de forma única por meio de seus índices.

$x[1][4] = 5$; \rightarrow atribui o valor 5 à posição identificada pelos índices 1 (2ª linha) e 4 (5ª coluna).

variável x

índice linhas da matriz	0					
	1				5	
		0	1	2	3	4

índice ou posições da matriz

$x[3][2] = 'D'$ \rightarrow Atribui a letra D à posição identificada pelos índices 3 (4ª linha) e 2 (3ª coluna).

variável x

índice linhas da matriz	0					
	1					
	2					
	3					D
		0	1	2	3	4

índice ou posições da matriz

Preenchendo uma matriz:

Preencher uma matriz significa percorrer todos os seus elementos, atribuindo-lhes um valor. Este valor pode ser recebido do usuário, por meio do teclado, ou pode ser gerado pelo programa.

No exemplo a seguir, todos os elementos de uma matriz bidimensional são percorridos, atribuindo-lhes valores inteiros digitados pelo usuário e capturados pelo método **nextInt()** da classe Scanner.

```

package com.mycompany.matrizexemplo1;

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

public class MatrizExemplo1 {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int[][] mtz = new int[7][3];
        int i, j;

        try {
            for (i = 0; i < 7; i++) {
                for (j = 0; j < 3; j++) {
                    System.out.print("Digite o valor para a matriz: ");
                    mtz[i][j] = sc.nextInt();
                }
            }
        } catch (InputMismatchException e) {
            System.out.println("Erro de digitação!");
        }
    }
}

```

Como a matriz possui sete linhas e três colunas, o **for** externo deve variar de 0 a 6 (percorrendo, assim, as sete linhas da matriz) e o **for** interno deve variar de 0 a 2 (percorrendo, assim, as três colunas da matriz).

Mostrando os elementos de uma matriz:

Pode-se, também, percorrer todos os elementos da matriz, acessando o seu conteúdo. No exemplo que se segue, são mostrados todos os elementos de uma matriz contendo dez linhas e seis colunas. Observe que são usados dois índices, **i** e **j**. Estes índices estão atrelados a estruturas de repetição que mantêm a variação de ambos dentro dos intervalos permitidos, ou seja, o índice **i**, que representa as linhas, varia de 0 a 9 e o índice **j**, que representa as colunas, varia de 0 a 5.

```

for (i = 0; i < 10; i++) {
    for (j = 0; j < 6; j++) {
        System.out.print(mtz[i][j]);
    }
}

```

```

/*
Faça um programa que preencha uma matriz M(2 x 2), calcule e mostre a matriz
R, resultante da multiplicação dos elementos de M pelo seu maior elemento.
*/

```

```

package com.mycompany.matriz;

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

public class Matriz {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int[][] m = new int[2][2];
        int[][] r = new int[2][2];
        int i, j, maior;

        try {

            // preenchendo a matriz com a digitação pelo usuario
            for (i = 0; i < 2; i++) {
                for (j = 0; j < 2; j++) {
                    System.out.print("Digite o valor da matriz: ");
                    m[i][j] = sc.nextInt();
                }
            }

            // buscando o maior valor da matriz digitada pelo usuario
            maior = 0;
            for (i = 0; i < 2; i++) {
                for (j = 0; j < 2; j++) {
                    if (m[i][j] > maior)
                        maior = m[i][j];
                }
            }

            // multiplicando a matriz m pelo maior valor
            // e colocado na matriz R
            for (i = 0; i < 2; i++) {
                for (j = 0; j < 2; j++) {
                    r[i][j] = m[i][j] * maior;
                }
            }

            // mostrando a matriz r, matriz resultante
            for (i = 0; i < 2; i++) {
                for (j = 0; j < 2; j++) {
                    System.out.print(r[i][j] + " ");
                }
                System.out.println(" ");
            }

        } catch (InputMismatchException e) {

            System.out.println("Erro de digitação!");
        }

    }
}

```

Saida:

```
Digite o valor da matriz: 1
Digite o valor da matriz: 5
Digite o valor da matriz: 9
Digite o valor da matriz: 7
9 45
81 63
```

A **Engenharia de Software** surgiu no final da década de 60 para tentar solucionar os problemas gerados pela “**Crise do Software**”, no entanto, várias técnicas que foram desenvolvidas nos anos 70 e 80 não conseguiram resolver os problemas de produtividade e qualidade de software.

Isso veio a melhorar por volta dos anos 90 onde, empresas de grande porte e com sistema considerados de grande quantidade de informação e informatização, mais o aparecimento da Internet, fez com que cada vez mais o computador se tornasse uma ferramenta essencial de trabalho na vida das pessoas, assim, fez com que os programas ficassem cada dia mais poderosos, com mais recursos e consumindo mais recursos do computador que, forçou um aperfeiçoamento de alteração e expansão de sistemas e interação com outros sistemas para troca de dados, portabilidade entre diversas plataformas.

Dessa forma, verificou-se que o reuso de partes de sistemas para alavancar a produção de software se tornou uma peça-chave nesse contexto, ou seja, reduzindo as etapas de desenvolvimento de software. Uma das técnicas que a programação começou a oferecer foi a **Programação orientada a objetos**, fazendo com que grandes diferenciais da programação orientada a objetos em relação a outros paradigmas de programação que também permitem a definição de estruturas e, operações sobre essas estruturas está no conceito de:

- **herança**, mecanismo através do qual definições existentes podem ser facilmente estendidas. Juntamente com a **herança** deve ser enfatizada a importância do;
- **polimorfismo**, que permite selecionar funcionalidades que um programa irá utilizar de forma dinâmica, durante sua execução.

Esse paradigma de programação está ancorado basicamente em algumas definições tais como:

- ✓ classes;
- ✓ objetos;
- ✓ herança e;
- ✓ polimorfismo.

Classes

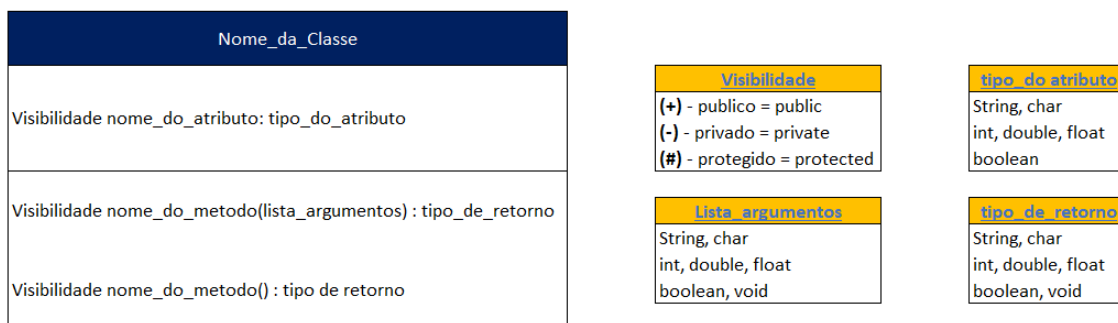
A palavra classe vem da taxonomia da biologia. Todos os seres vivos de uma mesma classe biológica têm uma série de atributos e comportamentos em comum, mas não são iguais, podem variar nos valores desses atributos e como realizam esses comportamentos, ou seja, uma classe é um **gabarito** para a **definição de objetos**. Através da definição de uma classe, descreve-se que

propriedades (**atributos**) que o objeto terá. Além dos atributos, a definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe que podem ser descritas através dos métodos.

Métodos

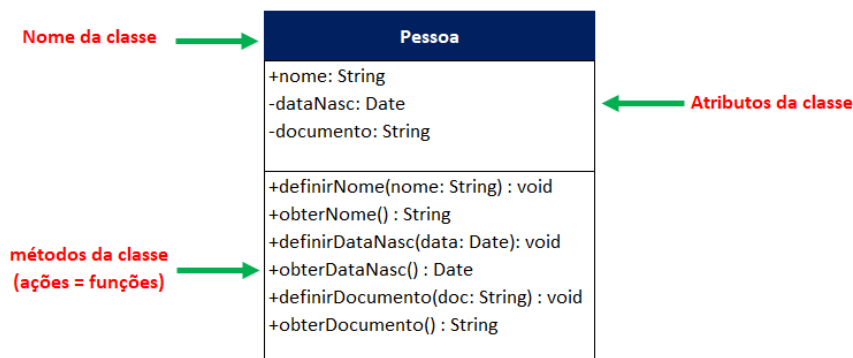
Um método nada mais é que o equivalente a um **procedimento** ou **função**, com a restrição que ele manipula apenas suas variáveis locais e os atributos que foram definidos para a classe. Uma vez que estejam definidas quais serão as classes que irão compor uma aplicação, assim como qual deve ser sua estrutura interna e comportamento, é possível criar essas classes em Java.

Na *Unified Modeling Language* (UML) – que veremos a seguir, a representação para uma classe no diagrama de classes é tipicamente expressa na forma **gráfica**, como mostrado na Figura a seguir.



Como se observa nessa figura, a especificação de uma classe é composta por três regiões:

- I. nome da classe;
- II. conjunto de atributos da classe e;
- III. conjunto de métodos da classe (funções).



O nome da **classe** é um identificador para a **classe**, que permite referenciá-la posteriormente, por exemplo, no momento da criação de um **objeto**, assim, o conjunto de **atributos** descreve as propriedades da classe e cada **atributo** é identificado por um nome e tem um **tipo** associado.

Os **métodos** definem as funcionalidades da classe, ou seja, o que será possível fazer com objetos dessa classe e cada **método** é especificado por uma **assinatura**, composta por:

1. Identificador para o **método** (o nome do método);
2. Tipo para o valor de retorno e;
3. Sua lista de argumentos, sendo cada argumento identificado por seu tipo e nome.

O modificador de visibilidade pode estar presente tanto para atributos como para métodos. Em princípio, três categorias de visibilidade podem ser definidas:

<u>publico</u>	Denotado em UML pelo símbolo +, assim, o atributo ou método de um objeto dessa classe pode ser acessado por qualquer outro objeto.
<u>privado</u>	Denotado em UML pelo símbolo -, assim, o atributo ou método de um objeto dessa classe não pode ser acessado por nenhum outro objeto.
<u>protegido</u>	Denotado em UML pelo símbolo #, assim, o atributo ou método de um objeto dessa classe poderá ser acessado apenas por objetos de classes que sejam derivadas dessa através do mecanismo de herança.

Objetos

Objetos são instâncias de classes. É através deles que (**praticamente**) todo o processamento ocorre em sistemas programados com linguagens de programação orientadas a objetos.

O uso racional de **objetos**, obedecendo aos princípios associados à sua definição conforme estabelecido no paradigma de desenvolvimento orientado a objetos, é a chave para o desenvolvimento de sistemas complexos e eficientes.

Um **objeto** é um elemento que representa, no domínio da solução, alguma entidade (**abstrata ou concreta**) do domínio de interesse do problema sob análise.

Objetos similares são agrupados em classes. Quando se cria um **objeto**, esse **objeto** adquire um espaço em memória para armazenar seu estado (**os valores de seu conjunto de atributos, definidos pela classe**) e um conjunto de operações que podem ser aplicadas ao objeto (o conjunto de métodos definidos pela classe).

Um programa orientado a objetos é composto por um conjunto de objetos que interagem através de “**trocas de mensagens**”. Na prática, essa troca de mensagens traduz-se na aplicação de métodos a objetos.

Exemplo-1:

Considere um **programa para um banco**, é bem fácil perceber que uma entidade extremamente importante para o nosso sistema é a **conta**. Nossa ideia aqui é generalizarmos alguma informação, juntamente com **funcionalidades** que toda **conta** deve ter.

O que toda conta tem e é importante para o desenvolvimento do sistema do banco:

1. número da conta;
2. nome do titular da conta e;
3. saldo.

O que toda conta faz e é importante, isto é, o que gostaríamos de "**pedir à conta**":

- sacar uma certa quantidade de valores;
- depositar uma certa quantidade;
- imprimir um extrato das transações da conta, nome do titular, número da conta e saldo.

Com isso, temos o projeto de uma conta bancária. Podemos pegar esse projeto e acessar seu saldo, correto?

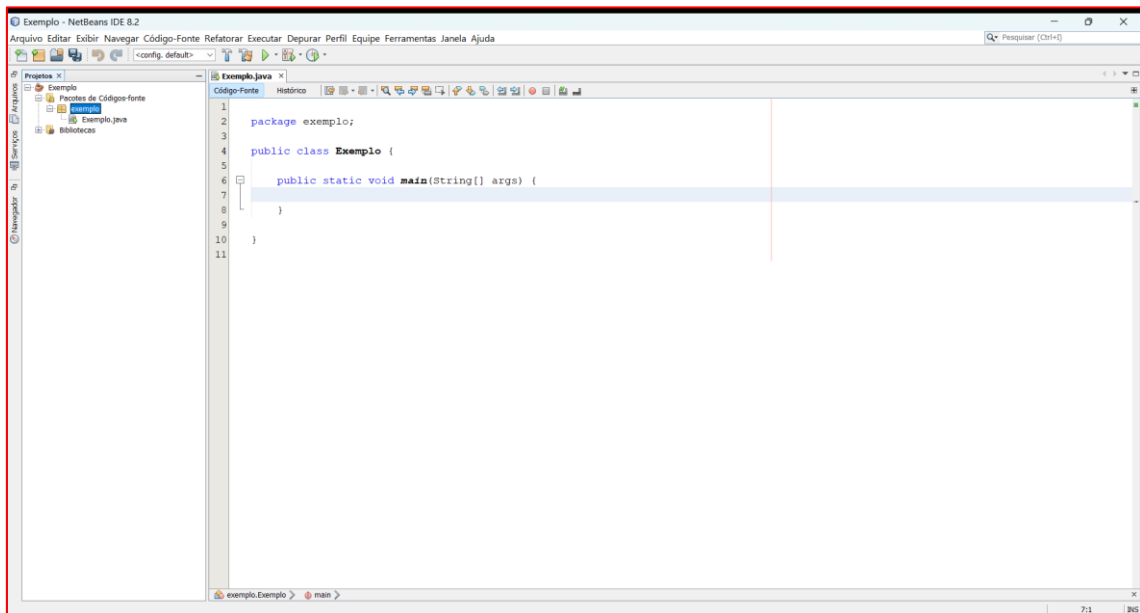
O que temos ainda é o projeto. Antes, precisamos construir uma conta, para poder acessar o que ela tem, e pedir a ela que faça algo.

Criando a classe:

Conta	
+nroConta: String +nomeTitular: String +saldo: double	atributos
+Sacar(valor: double) : boolean +Depositar(valor: double) : void +Extrato() : void	

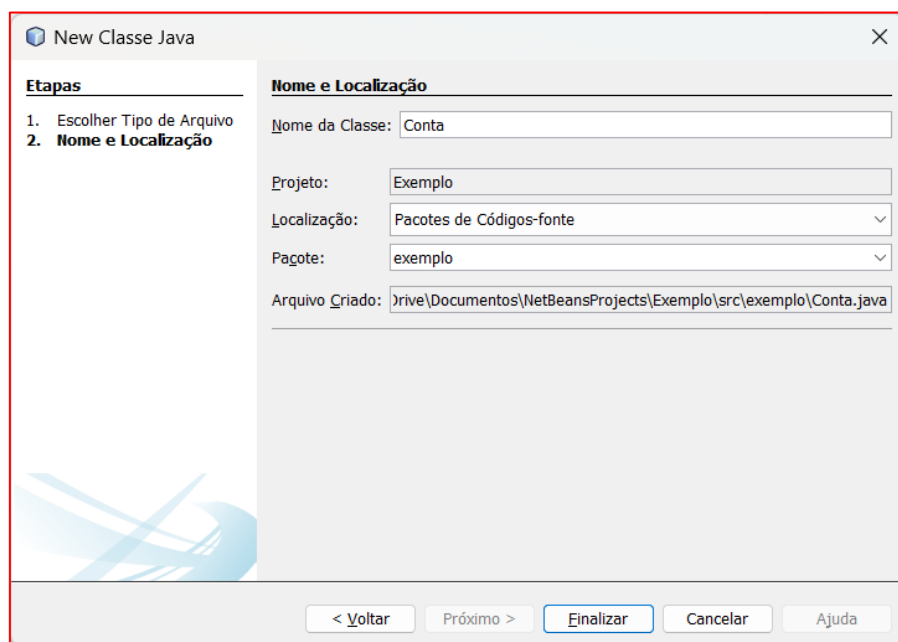
Agora, vamos transportar esse diagrama para o código no NetBeans:

1. Criar um projeto **Exemplo**

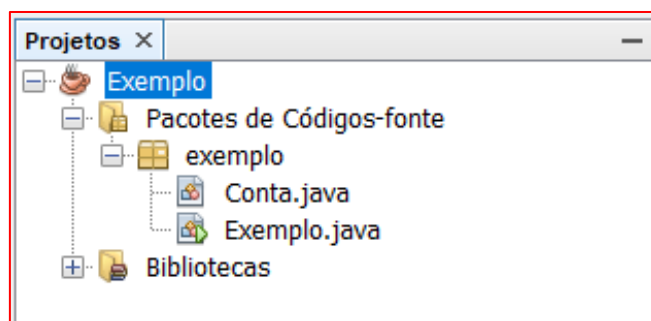


2. Com o pacote do projeto conforme figura anterior, após criar o projeto Exemplo, vamos criar a classe **conta**

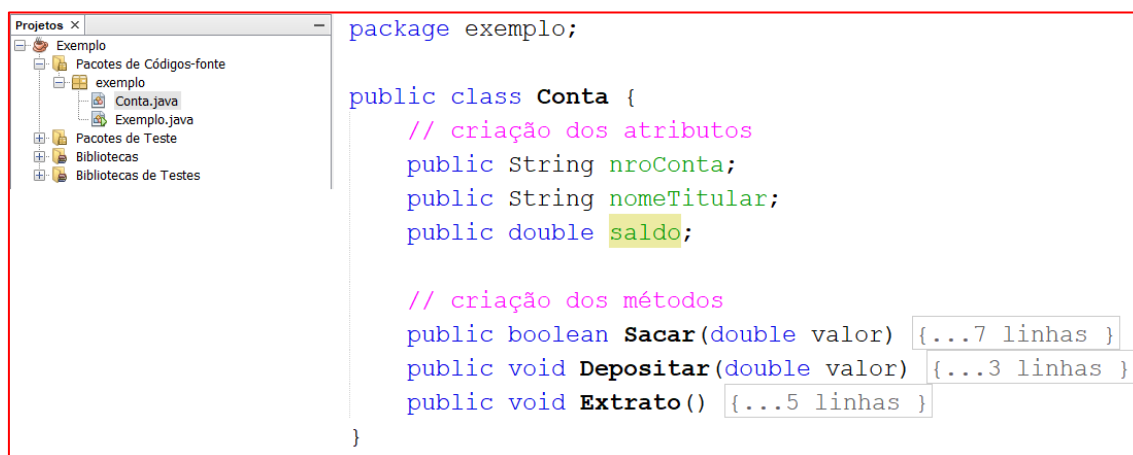
2.1. Com o botão direito do mouse no pacote do projeto, selecione Novo ou New, em seguida clique em Classe Java ou Java Class e será apresentado a tela abaixo, onde vamos digitar o nome da classe **Conta**.



3. Observando a estrutura do projeto no NetBeans, temos:



4. Agora podemos clicar no arquivo Conta.java e passar do diagrama **Conta** criado para o código.



Dentro da classe, também declararemos o que cada conta faz e como isto é feito - **os comportamentos que cada classe tem, isto é, o que ela faz**. Por exemplo, de que maneira que uma Conta saca dinheiro? Especificaremos isso dentro da própria classe **Conta**, e não em um local desatrelado das informações da própria Conta. É por isso que essas "**funções**" são chamadas de métodos. Pois é a maneira de fazer uma operação com um objeto.

Queremos criar um método que **saca** uma determinada quantidade e tem como retorno (verdadeiro – sim, há saldo na conta ou falso – não, há saldo suficiente para o saque), informação para quem acionar esse método:

```
public boolean Sacar(double valor) {
    if (valor >= this.saldo) {
        this.saldo -= valor;
        return true;
    }
    return false;
}
```

Quando alguém pedir para **sacar**, ele também vai dizer quanto quer sacar. Por isso precisamos declarar o método com algo dentro dos parênteses - **o que vai aí dentro é chamado de argumento do método (ou parâmetro)**. Essa variável é uma variável comum, chamada também de temporária ou local, pois, ao final da execução desse método, ela deixa de existir.

Usamos a palavra-chave **this** para mostrar que esse é um atributo, e não uma simples variável.

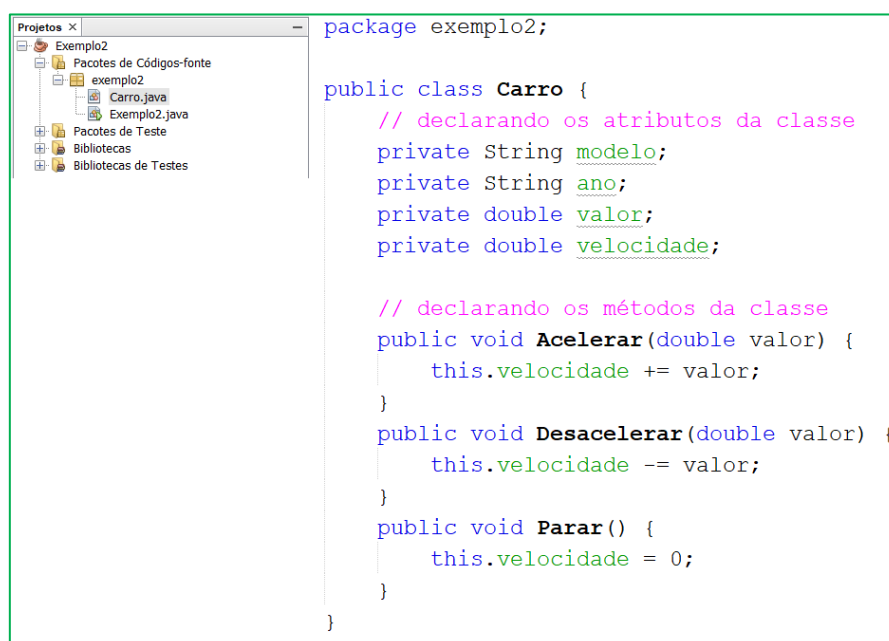
this.saldo -= valor;

Observe que não usamos uma variável auxiliar e, além disso, usamos a abreviação **-=** para deixar o método bem simples. O **-=** subtrai a quantidade ao valor antigo do saldo e guarda no próprio saldo, o valor resultante.

Da mesma forma, temos o método para depositar alguma quantia:

```
public void Depositar(double valor) {
    this.saldo += valor;
}
public void Extrato() {
    System.out.println("Nro da conta: " + this.nroConta);
    System.out.println("Nome do titular: " + this.nomeTitular);
    System.out.println("Sado da conta: " + this.saldo);
}
```

Exemplo-2:



```
package exemplo2;

public class Carro {
    // declarando os atributos da classe
    private String modelo;
    private String ano;
    private double valor;
    private double velocidade;

    // declarando os métodos da classe
    public void Acelerar(double valor) {
        this.velocidade += valor;
    }
    public void Desacelerar(double valor) {
        this.velocidade -= valor;
    }
    public void Parar() {
        this.velocidade = 0;
    }
}
```

Exemplo prático-1:

O preço, ao consumidor, de um carro novo e a soma do custo de fábrica com a porcentagem do distribuidor e com os impostos, ambos aplicados ao custo de fábrica. As porcentagens encontram-se na tabela a seguir. Faça um projeto em Java que receba o custo de fábrica de um carro e mostre o preço ao consumidor. Para isso, crie uma **classe fabrica** que apresente os percentuais do distribuidor e a porcentagem dos impostos para o carro.

CUSTO DE FABRICA	% DISTRIBUIDOR	% IMPOSTOS
Até R\$ 21.000,00	5	isento
Entre R\$ 21.000,00 e R\$ 52.000,00	10	15
Acima de R\$ 52.000,00	15	20

```
package exemplo;

public class Fabrica {
    // declarando os atributos da classe
    public double custo;
    public double custoFabrica;
    public double custoImpostos;

    // metodos da classe
    public double valorDistribuidor() {

        if (this.custo <= 21000) {
            this.custoFabrica = this.custo * 0.05;
        } else if (this.custo > 21000 && this.custo <= 52000) {
            this.custoFabrica = this.custo * 0.10;
        } else {
            this.custoFabrica = this.custo * 0.15;
        }
        return this.custoFabrica;
    }

    public double valorImpostos() {

        if (this.custo <= 21000) {
            this.custoImpostos = 0;
        } else if (this.custo > 21000 && this.custo <= 52000) {
            this.custoImpostos = this.custo * 0.15;
        } else {
            this.custoImpostos = this.custo * 0.20;
        }
        return this.custoImpostos;
    }
}
```

```

package exemplo;

public class Exemplo {

    public static void main(String[] args) {
        Fabrica fabrica = new Fabrica();
        int i;

        // vamos executar os valores para 10 carros
        for (i = 0; i < 10; i++) {
            // valor aleatorio para o carro
            fabrica.custo = 1 + Math.random() * 60000;

            // estabelece o valor do distribuidor
            fabrica.valorDistribuidor();

            // estabelece o valor do imposto
            fabrica.valorImpostos();

            // Relatorio final
            System.out.println("Custo do carro: R$ " + String.format("%.2f", fabrica.custo));
            System.out.println("Custo do distribuidor: R$ " + String.format("%.2f", fabrica.custoFabrica));
            System.out.println("Custo do imposto: R$ " + String.format("%.2f", fabrica.custoImpostos));
            System.out.println("");
        }
    }
}

```

Exemplo prático-2:

Faça um projeto em Java que receba o valor de uma dívida e mostre uma tabela com os seguintes dados:

- Valor da dívida;
- Valor dos juros;
- Quantidade de parcelas e;
- Valor da parcela.

Os juros e a quantidade de parcelas seguem a tabela:

QUANTIDADE DE PARCELAS	% DE JUROS SOBRE O VALOR INICIAL DA DÍVIDA
1	0
3	10
6	15
9	20
12	25

Exemplo de Saída:

VALOR DA DÍVIDA	VALOR DOS JUROS	QUANTIDADE DE PARCELAS	VALOR DA PARCELA
R\$ 1.000,00	0	1	R\$ 1.000,00
R\$ 1.100,00	100	3	R\$ 366,67
R\$ 1.150,00	150	6	R\$ 191,67

```

package exemplo;

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

public class Exemplo {

    public static void main(String[] args) {
        CalculaDivida calc = new CalculaDivida();
        Scanner sc = new Scanner(System.in);
        double valorDivida;

        try {

            System.out.println("");
            System.out.print("Digite o valor da dívida: ");
            valorDivida = sc.nextDouble();
            calc.calcular(valorDivida);
            calc.montaTabela();

        } catch (InputMismatchException ex) {
            System.out.println("Erro de digitação!");
        }

    }
}

```

```

package exemplo;
public class CalculaDivida {
    // declarando os atributos
    public double [][] tabela = new double[5][4];

    // declarando os metodos da classe
    public void calcular(double valor) {
        int i, parc;
        double indice;

        this.tabela[0][0] = valor;
        this.tabela[0][1] = 0;
        this.tabela[0][2] = 1;
        this.tabela[0][3] = valor;
        indice = 0.10;
        parc = 3;

        for (i = 1; i < 5; i++) {
            this.tabela[i][0] = valor + (valor * indice);
            this.tabela[i][1] = valor * indice;
            this.tabela[i][2] = parc;
            this.tabela[i][3] = (valor + (valor * indice)) / parc;
            indice += 0.05;
            parc += 3;
        }

    }

    public void montaTabela() {
        int i;

        System.out.println(" ");
        System.out.println("VALOR DÉVIDA \t VALOR DOS JUROS \t QTDE PARCELAS \t VALOR PARCELA");
        for (i = 0; i < 5; i++) {
            System.out.print("R$ " + String.format("%.2f", this.tabela[i][0]) + "\t ");
            System.out.print("R$ " + String.format("%.2f", this.tabela[i][1]) + "\t\t\t");
            System.out.print((int)this.tabela[i][2] + "\t ");
            System.out.print("R$ " + String.format("%.2f", this.tabela[i][3]));
            System.out.println("");
        }
        System.out.println("\n");
    }
}

```


Orientação a objetos

Uma vez estudados os **fundamentos** e **sintaxe básicas** para a construção de **objetos**, e que devem ser muito bem assimilados, o objeto de estudo, em seguida, são os **principais conceitos de orientação a objetos**, detalhando suas **características** e **aplicabilidade**.

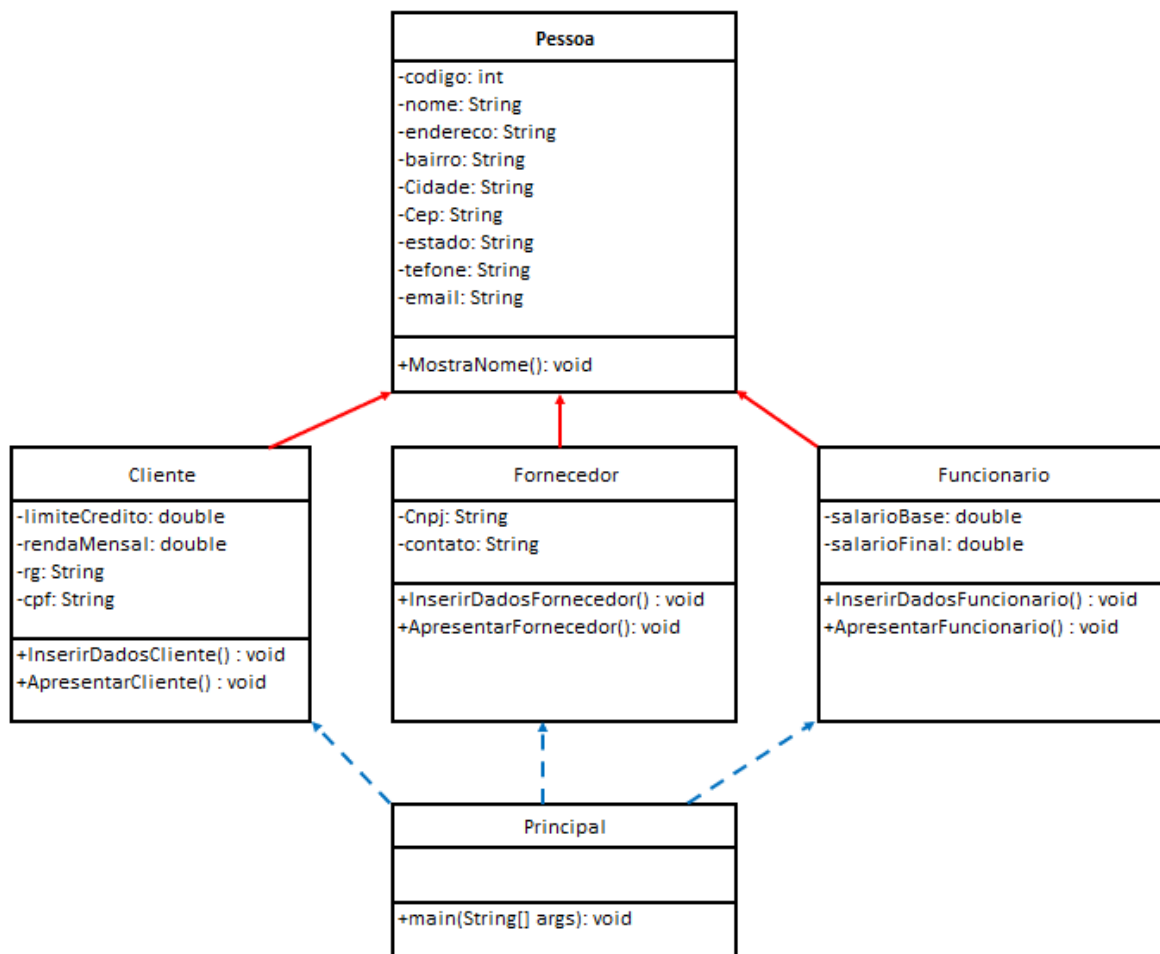
Herança

A herança é um conceito amplamente utilizado em linguagens orientadas a objetos. Além de vantagens facilmente identificadas, como a **reutilização** e **organização** de códigos, a herança também é a base para outros conceitos, como a **sobrescrita de métodos**, **classes e métodos abstratos** e **polimorfismo**. Tais conceitos são fundamentais para a modelagem de sistemas mais robustos.

Durante a análise dos requisitos de um sistema (**solicitações que o sistema deverá atender**), podemos destacar os atributos ou os métodos comuns a um grupo de classes e concentrá-los em uma única classe (**processo conhecido como generalização**). Da mesma forma, é possível identificar o que é pertinente somente a determinada classe (**conhecido como especificação**). A primeira vantagem dessa organização é **evitar a duplicidade de código** (ter o mesmo trecho de código em lugares diferentes do sistema), o que traz maior **agilidade** e **confiabilidade** na manutenção e expansão do sistema.

Chamamos de **superclasses** essas classes que concentram atributos e métodos comuns que podem ser reutilizados (**herdados**) e de **subclasse** aquelas que reaproveitam (**herdam**) esses recursos.

Exemplo: Vejamos um exemplo, observe as definições das classes **Cliente**, **Fornecedor** e **Funcionário** utilizados no diagrama abaixo.




A classe **Pessoa** contém nove atributos e um método, os quais são comuns para clientes, fornecedores e funcionários e, portanto, deveriam constar nas classes **Cliente**, **Fornecedor** e **Funcionário**. Sem o recurso da herança, teríamos que replicar esses atributos e métodos nas três classes, procedimento desaconselhável em qualquer linguagem de programação, por trazer complexidades extras na manutenção e expansão dos sistemas. Por exemplo, vamos considerar um método para emissão de correspondência que foi atualizado para começar a gerar um histórico de remessas. Tal atualização deveria ser feita nas três classes envolvidas e, caso uma delas não fosse realizada, a atualização do controle de remessas (geração de histórico) ficaria inconsistente.

No modelo acima, a classe **Pessoa** foi definida como uma **superclasse** e as classes **Cliente**, **Fornecedor** e **Funcionário**, como suas **subclasses**. Do ponto de vista da funcionalidade, tudo o que foi definido na **superclasse** (atributos e métodos) será **herdado** pelas suas subclasses. Ou seja, um **objeto instanciado a partir da classe Cliente possui 13 atributos**. São eles: “**nome, endereço, bairro, cidade, estado, telefone e email**” declarados na superclasse **Pessoa**, além de “**limiteCredito, rendaMensal, rg e cpf**”, na subclasse **Cliente**. Há ainda três métodos,

nos quais “**MostrarNome**” foi definido na classe **Pessoa** e “**InserirDadosCliente** e **ApresentarCleinte**” foram definidos na classe **Cliente**. Na utilização desses atributos e métodos para um objeto do tipo **Cliente**, fica transparente o local onde cada um foi declarado ou definido.

Para estabelecer a herança em relação à codificação, as superclasses continuam com a mesma estrutura de uma classe comum. Já as subclasses recebem as seguintes definições:

```
public class Cliente extends Pessoa{...}
```



aqui está a “herança” em comando no Java.

O comando **extends** é o responsável por estabelecer a herança. É inserido na abertura da subclasse e indica o nome da superclasse, criando vínculo entre elas.

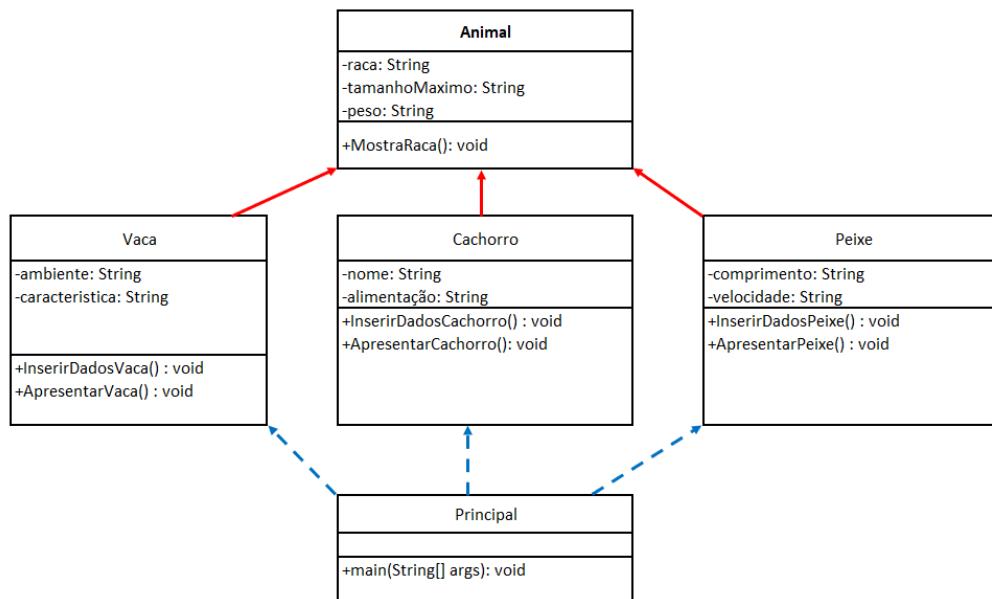
Construtores

Os construtores estão diretamente relacionados à inicialização dos atributos de uma classe. Partindo desse princípio e considerando o nosso exemplo, um objeto do tipo cliente possui todos os atributos declarados na sua **superclasse** (**Pessoa**) mais os declarados na classe **Cliente**. Portanto, o **construtor de uma subclasse deve estar preparado para inicializar os atributos** herdados e os declarados na própria classe.

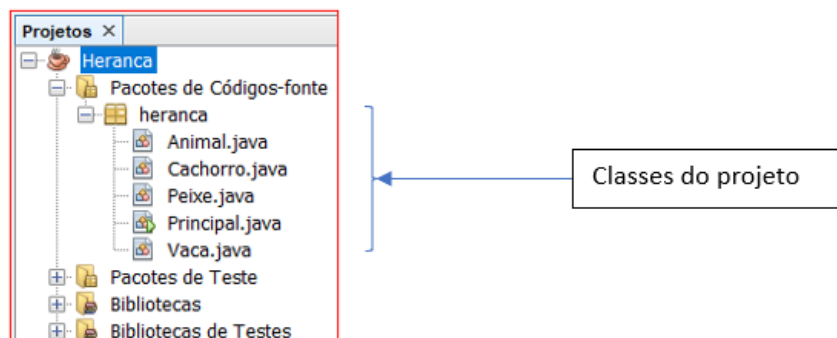
No construtor que recebe parâmetros (**aquele que inicializa os atributos com algum valor**), utilizamos o método **super** () para invocar o construtor da superclasse. Isso porque já foi definida nele a forma como esses atributos serão inicializados (**reutilizando o construtor já existente na superclasse**), restando apenas inicializar os atributos na subclasse.

Para acessar os atributos da **superclasse**, obrigatoriamente, devemos utilizar seus métodos de acesso (**getters e setters**), ao contrário do atributos instanciados na própria classe, que podem ser acessados diretamente. Porém, para garantir o conceito de **encapsulamento** e usufruir de seus benefícios (**segurança, manutenibilidade etc.**), sempre devemos criar e utilizar os métodos **getters** e **setters** para todos os atributos. Em relação ao nosso exemplo, ele se aplica às classes **Fornecedor** e **Funcionário**.

Exemplo-1: Vamos desenvolver o projeto de acordo com o diagrama abaixo.



Estrutura do projeto:



```
package heranca;

public class Animal {
    // declarando os atributos da classe
    private String raca;
    private String tamanhoMaximo;
    private String peso;

    // metodo da classe Animal
    public void MostrarRaca() {
        System.out.println("Raça do animal: " + this.raca);
    }

    // metodos get's e set's da classe
    public String getRaca() {
        return raca;
    }
    public void setRaca(String raca) {
        this.raca = raca;
    }
    public String getTamanhoMaximo() {
        return tamanhoMaximo;
    }
    public void setTamanhoMaximo(String tamanhoMaximo) {
        this.tamanhoMaximo = tamanhoMaximo;
    }
    public String getPeso() {
        return peso;
    }
    public void setPeso(String peso) {
        this.peso = peso;
    }
}
```

```
package heranca;

public class Vaca extends Animal {
    // declarando os atributos da classe
    private String ambiente;
    private String caracteristica;

    // declarando os metodos da classe
    public void InserirDadosVaca() {
        super.setRaca("Nelore");
        super.setPeso("600kg");
        super.setTamanhoMaximo("média 165cm de comprimento e 155cm de altura");
        this.ambiente = "Terra";
        this.caracteristica = "Fêmeas apresentam musculatura menos desenvolvida";
    }
    public void ApresentarVaca() {
        System.out.println("Raça da vaca....: " + super.getRaca());
        System.out.println("Tamanho Maximo..: " + super.getTamanhoMaximo());
        System.out.println("Peso da vaca....: " + super.getPeso());
        System.out.println("Ambiente da vaca: " + this.ambiente);
        System.out.println("Caracteristica..: " + this.caracteristica);
    }
}
```

```
package heranca;

public class Cachorro extends Animal {
    // declarando os atributos da classe
    private String nome;
    private String alimentacao;

    // declarando os metodos da classe
    public void InserirDadosVaca() {
        super.setRaca("Golden");
        super.setPeso("50kg");
        super.setTamanhoMaximo("média 55cm de comprimento e 45cm de altura");
        this.nome = "Peter";
        this.alimentacao = "Ração Premier";
    }
    public void ApresentarVaca() {
        System.out.println("Raça.....: " + super.getRaca());
        System.out.println("Tamanho Maximo..: " + super.getTamanhoMaximo());
        System.out.println("Peso.....: " + super.getPeso());
        System.out.println("Nome.....: " + this.nome);
        System.out.println("Alimentação.....: " + this.alimentacao);
    }
}
```

```
package heranca;

public class Peixe extends Animal {
    // declarando os atributos da classe
    private String comprimento;
    private String velocidade;

    // declarando os metodos da classe
    public void InserirDadosVaca() {
        super.setRaca("Tubarao-branco");
        super.setPeso("2,5 toneladas");
        super.setTamanhoMaximo("8 metros");
        this.comprimento = "em média de 7 metros de comprimento";
        this.velocidade = "50 m/s";
    }
    public void ApresentarVaca() {
        System.out.println("Raça.....: " + super.getRaca());
        System.out.println("Tamanho Maximo...: " + super.getTamanhoMaximo());
        System.out.println("Peso.....: " + super.getPeso());
        System.out.println("Comprimento.....: " + this.comprimento);
        System.out.println("Velocidade.....: " + this.velocidade);
    }
}
```

```
package heranca;

public class Principal {

    public static void main(String[] args) {
        // fazendo as instancias das classes (subclasses)
        Vaca vaca = new Vaca();
        Peixe peixe = new Peixe();
        Cachorro dog = new Cachorro();

        vaca.InserirDadosVaca();
        vaca.ApresentarVaca();
        System.out.println(""); // pula linha em branco

        peixe.InserirDadosVaca();
        peixe.ApresentarVaca();
        System.out.println(""); // pula linha em branco

        dog.InserirDadosVaca();
        dog.ApresentarVaca();
        System.out.println(""); // pula linha em branco
    }
}
```

Resultado:

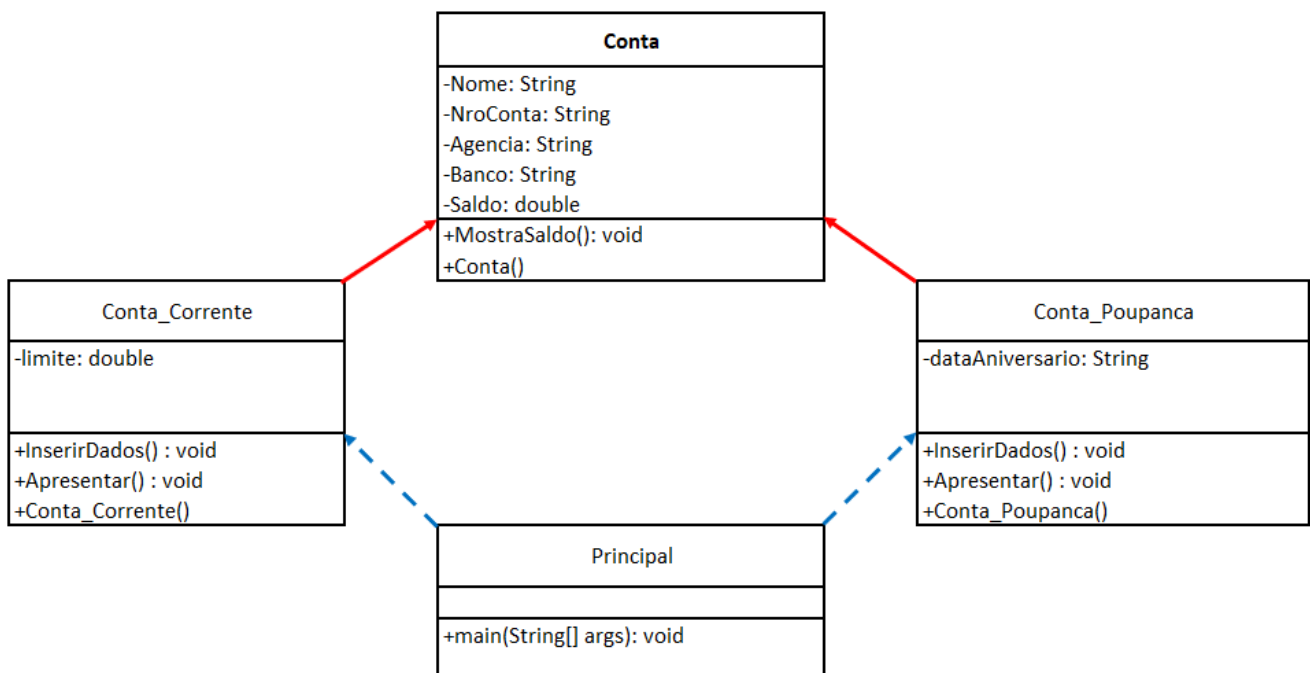
```

Raça da vaca.....: Nelore
Tamanho Maximo...: média 165cm de comprimento e 155cm de altura
Peso da vaca.....: 600kg
Ambiente da vaca: Terra
Caracteristica...: Fêmeas apresentam musculatura menos desenvolvida

Raça.....: Tubarao-branco
Tamanho Maximo...: 8 metros
Peso.....: 2,5 toneladas
Comprimento.....: em média de 7 metros de comprimento
Velocidade.....: 50 m/s

Raça.....: Golden
Tamanho Maximo...: média 55cm de comprimento e 45cm de altura
Peso.....: 50kg
Nome.....: Peter
Alimentação.....: Ração Premier
  
```

Exemplo-2: Vamos desenvolver o projeto de acordo com o diagrama abaixo, lembrando que agora temos o construtor da classe.




```
package herancacomconstrutor;

public class Conta {
    // declarando os atributos da classe
    private String Nome;
    private String NroConta;
    private String Agencia;
    private String Banco;
    private final double Saldo;

    // construtor da classe
    public Conta() {
        this.Nome = "nome do cliente";
        this.NroConta = "nro da conta";
        this.Agencia = "Agência";
        this.Banco = "Banco";
        this.Saldo = 0;
    }

    // metodo da classe
    public void MostrarSaldo() {
        System.out.println("Saldo: R$ " + String.format("%.2f", this.Saldo));
    }

    // metodos get's e set's
    public String getNome() {
        return Nome;
    }
    public void setNome(String Nome) {
        this.Nome = Nome;
    }
    public String getNroConta() {
        return NroConta;
    }
    public void setNroConta(String NroConta) {
        this.NroConta = NroConta;
    }
    public String getAgencia() {
        return Agencia;
    }
    public void setAgencia(String Agencia) {
        this.Agencia = Agencia;
    }
    public String getBanco() {
        return Banco;
    }
    public void setBanco(String Banco) {
        this.Banco = Banco;
    }
}
```

```
package herancacomconstrutor;

import javax.swing.JOptionPane;

public class Conta_Poupanca extends Conta {
    // declarando os atributos da classe
    private String dataAniversario;

    // construtor da classe
    public Conta_Poupanca() {
        super(); // chamando o construtor da superclass
        this.dataAniversario = "15 do mês";
    }

    // metodo da classe
    public void InserirDados() {
        super.setNome("Marcos Antonio");
        super.setBanco("23 - Bradesco");
        super.setAgencia("1363-3");
        super.setNroConta("100.987-0");
    }

    public void Apresentar() {
        String aux = "\nDados da conta Poupança\n\n";

        aux += "Nome: " + super.getNome() + "\n";
        aux += "Banco: " + super.getBanco() + "\n";
        aux += "Agência: " + super.getAgencia() + "\n";
        aux += "Nro da Conta: " + super.getNroConta() + "\n";
        aux += "Limite da Conta: " + this.dataAniversario + "\n";

        JOptionPane.showMessageDialog(null, aux);
    }
}
```

Erro: correto Data Aniversário

```
package herancacomconstrutor;

import javax.swing.JOptionPane;

public class Conta_Corrente extends Conta {
    // declarando os atributos da classe
    private double limite;

    // construtor da classe
    public Conta_Corrente() {
        super(); // chamando o construtor da superclass
        this.limite = 0;
    }

    // metodo da classe
    public void InserirDados() {
        super.setNome("Carlos Eduardo Mattos");
        super.setBanco("001 - Banco do Brasil");
        super.setAgencia("0978-8");
        super.setNroConta("22.545-9");
    }

    public void Apresentar() {
        String aux = "\nDados da Conta Corrente\n\n";

        aux += "Nome: " + super.getNome() + "\n";
        aux += "Banco: " + super.getBanco() + "\n";
        aux += "Agência: " + super.getAgencia() + "\n";
        aux += "Nro da Conta: " + super.getNroConta() + "\n";
        aux += "Limite da Conta: " + this.limite + "\n";

        JOptionPane.showMessageDialog(null, aux);
    }
}
```

```

package herancacomconstrutor;

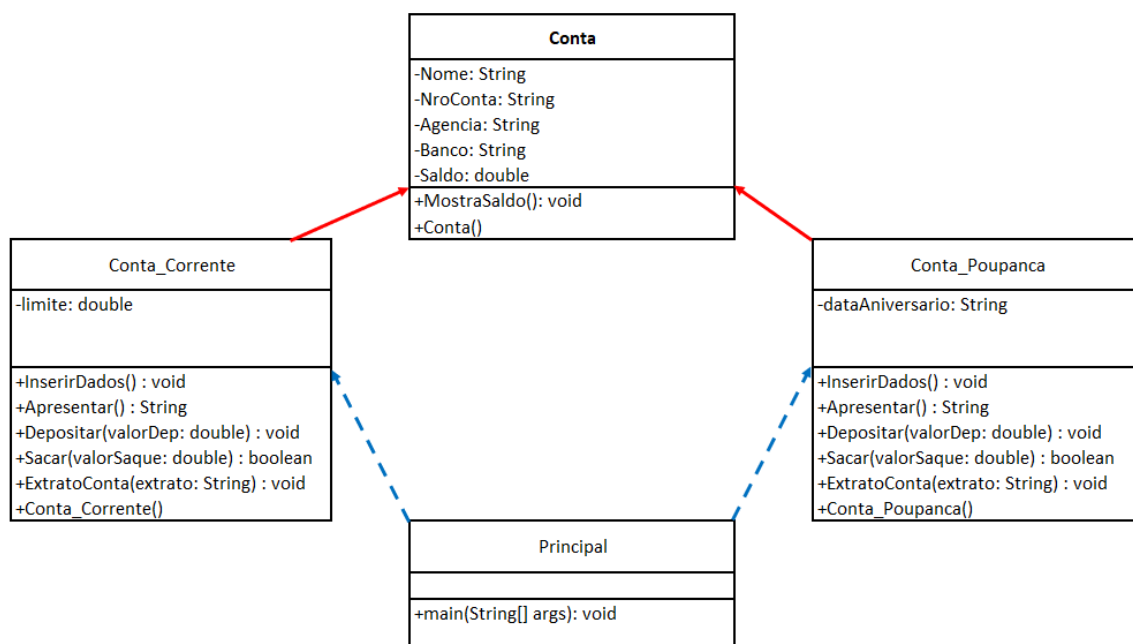
public class Principal {

    public static void main(String[] args) {
        Conta_Corrente conta = new Conta_Corrente();
        Conta_Poupanca poupanca = new Conta_Poupanca();

        conta.InserirDados();
        conta.Apresentar();

        poupanca.InserirDados();
        poupanca.Apresentar();
    }
}
    
```

Exemplo-3: Aproveitando o diagrama acima e modificando alguns itens, faça um projeto em Java que simule 20 transações de depósito e saques com as classes `Conta_Corrente` e `Conta_Poupanca`, ou seja, 10 transações para cada classe.



```
package herancacomconstrutor;

public class Conta {
    // declarando os atributos da classe
    private String Nome;
    private String NroConta;
    private String Agencia;
    private String Banco;
    private double Saldo;

    // construtor da classe
    public Conta() {
        this.Nome = "nome do cliente";
        this.NroConta = "nro da conta";
        this.Agencia = "Agência";
        this.Banco = "Banco";
        this.Saldo = 0;
    }

    // metodo da classe
    public void MostrarSaldo() {
        System.out.println("Saldo: R$ " + String.format("%.2f", this.Saldo));
    }

    // metodos get's e set's
    public String getNome() {
        return Nome;
    }
    public void setNome(String Nome) {
        this.Nome = Nome;
    }
    public String getNroConta() {
        return NroConta;
    }
    public void setNroConta(String NroConta) {
        this.NroConta = NroConta;
    }
    public String getAgencia() {
        return Agencia;
    }
    public void setAgencia(String Agencia) {
        this.Agencia = Agencia;
    }
    public String getBanco() {
        return Banco;
    }
    public void setBanco(String Banco) {
        this.Banco = Banco;
    }
    public double getSaldo() {
        return Saldo;
    }
    public void setSaldo(double Saldo) {
        this.Saldo += Saldo;
    }
}
```



```
package herancacomconstrutor;

import javax.swing.JOptionPane;

public class Conta_Corrente extends Conta {
    // declarando os atributos da classe
    // "final" pq não sofre alteração
    private final double limite;

    // construtor da classe
    public Conta_Corrente() {
        super(); // chamando o construtor da superclass
        this.limite = 100;
    }

    // metodo da classe
    public void InserirDados() {
        super.setNome("Carlos Eduardo Mattos");
        super.setBanco("001 - Banco do Brasil");
        super.setAgencia("0978-8");
        super.setNroConta("22.545-9");
    }

    public String Apresentar() {
        String aux = "\nDados da Conta Corrente\n";

        aux += "Nome: " + super.getNome() + "\n";
        aux += "Banco: " + super.getBanco() + "\n";
        aux += "Agência: " + super.getAgencia() + "\n";
        aux += "Nro da Conta: " + super.getNroConta() + "\n";
        aux += "Limite da Conta: " + this.limite + "\n";
        return aux;
    }

    public void Depositar(double valorDep) {
        // atribuindo o valorDep ao atributo da classe saldo
        this.setSaldo(valorDep);
    }

    public boolean Sacar(double valorSaque) {
        // buscando o saldo da conta corrente
        double saldo = super.getSaldo();
        // total de saldo + limite da conta
        double total = saldo + this.limite;

        if ( total >= valorSaque ) {
            return true; // sim, o cliente tem saldo para saque
        }
        return false; // não, o cliente não tem saldo para saque
    }

    public void ExtratoConta(String extrato) {
        String aux = "";

        aux += Apresentar() + "\n";
        aux += extrato + "\n";
        aux += "\nSaldo Final: R$ " + String.format("%.2f", super.getSaldo());
        JOptionPane.showMessageDialog(null, aux);
    }
}
```

```
package herancacomconstrutor;

import javax.swing.JOptionPane;

public class Conta_Poupanca extends Conta {
    // declarando os atributos da classe
    private String dataAniversario;

    // construtor da classe
    public Conta_Poupanca() {
        super(); // chamando o construtor da superclass
        this.dataAniversario = "15 do mês";
    }

    // metodo da classe
    public void InserirDados() {
        super.setNome("Marcos Antonio");
        super.setBanco("23 - Bradesco");
        super.setAgencia("1363-3");
        super.setNroConta("100.987-0");
    }

    public String Apresentar() {
        String aux = "\nDados da conta Poupança\n";

        aux += "Nome: " + super.getNome() + "\n";
        aux += "Banco: " + super.getBanco() + "\n";
        aux += "Agência: " + super.getAgencia() + "\n";
        aux += "Nro da Conta: " + super.getNroConta() + "\n";
        aux += "Data aniversario: " + this.dataAniversario + "\n";
        return aux;
    }

    public void Depositar(double valorDep) {
        // atribuindo o valorDep ao atributo da classe saldo
        this.setSaldo(valorDep);
    }

    public boolean Sacar(double valorSaque) {
        // buscando o saldo da conta corrente
        double saldo = super.getSaldo();
        // total de saldo + limite da conta
        double total = saldo;

        if ( total >= valorSaque ) {
            return true; // sim, o cliente tem saldo para saque
        }
        return false; // não, o cliente não tem saldo para saque
    }

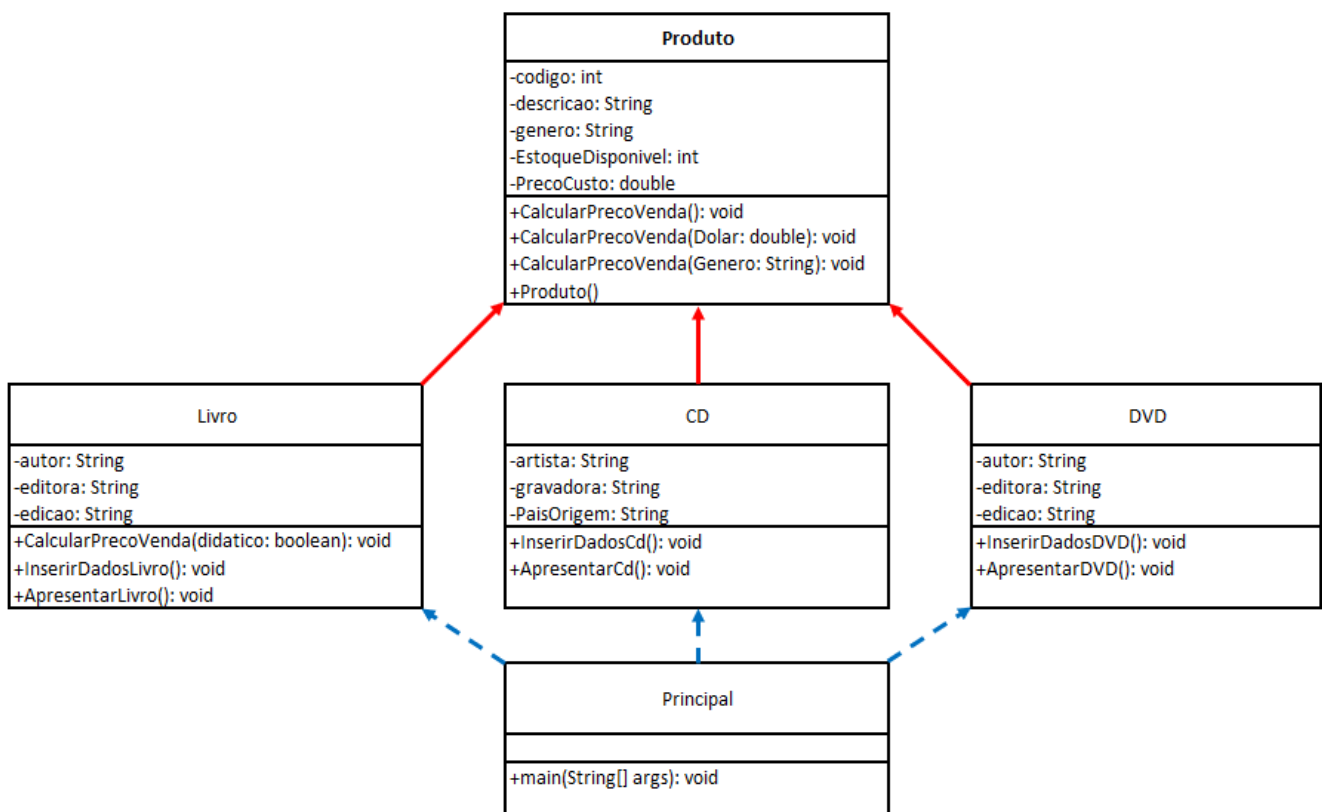
    public void ExtratoConta(String extrato) {
        String aux = "";

        aux += Apresentar() + "\n";
        aux += extrato + "\n";
        aux += "\nSaldo Final: R$ " + String.format("%.2f", super.getSaldo());
        JOptionPane.showMessageDialog(null, aux);
    }
}
```



```
public class Principal {  
  
    public static void main(String[] args) {  
        Conta_Corrente conta = new Conta_Corrente();  
        Conta_Poupanca poupanca = new Conta_Poupanca();  
        Random rand = new Random(); // Classe que gera nros aleatorios  
  
        // inserindo dados da conta poupança e conta corrente  
        poupanca.InserirDados();  
        conta.InserirDados();  
  
        // variaveis primitivas  
        int i;  
        double valor;  
        String transacaoPoup = "\nTransacao Poupanca\n"; // cabeçalho  
        String transacaoConta = "\nTransacao Conta Corrente\n"; // cabeçalho  
        String dados;  
  
        // simular 50 transações utilizando o for  
        for (i = 0; i < 20; i++) {  
            // transacao pares direciona para Conta corrente  
            if (i % 2 == 0) {  
  
                valor = rand.nextDouble() * 1000; // valor aleatorio  
                poupanca.Depositar(valor); // soma com o saldo  
                transacaoPoup += i + "-Deposito: R$ " + String.format("%.2f", valor) + "\n";  
  
                valor = rand.nextDouble() * 1000; // valor aleatorio  
                valor = valor * -1; // para deixar negativo  
                if (poupanca.Sacar(valor)) { // verifica se pode sacar  
                    poupanca.setSaldo(valor);  
                    transacaoPoup += i + "-Saque: R$ " + String.format("%.2f", valor) + "\n";  
                } else { // não tem saldo  
                    transacaoPoup += i + "-Saldo insuficiente" + "\n";  
                }  
  
            } else {  
  
                valor = rand.nextDouble() * 1000; // valor aleatorio  
                conta.Depositar(valor); // soma com o saldo  
                transacaoConta += i + "-Deposito: R$ " + String.format("%.2f", valor) + "\n";  
  
                valor = rand.nextDouble() * 1000; // valor aleatorio  
                valor = valor * -1; // para deixar negativo  
                if (conta.Sacar(valor)) { // verifica se pode sacar  
                    conta.setSaldo(valor);  
                    transacaoConta += i + "-Saque: R$ " + String.format("%.2f", valor) + "\n";  
                } else { // não tem saldo  
                    transacaoConta += i + "-Saldo insuficiente" + "\n";  
                }  
  
            }  
        }  
  
        dados = poupanca.Apresentar();  
        dados = transacaoPoup;  
        poupanca.ExtratoConta(dados);  
  
        dados = conta.Apresentar();  
        dados = transacaoConta;  
        conta.ExtratoConta(dados);  
    }  
}
```


A **programação** em sistemas desenvolvidos com **Java** é **distribuída** e **organizada em métodos**. Muitas vezes, os programadores se deparam com situações em que um método deve ser usado para finalidades semelhantes, mas com dados diferentes. Por exemplo, os produtos comercializados em uma livraria onde podemos ter Livros, CD's e DVD's, como representado no diagrama de classe abaixo:



O cálculo do preço de venda dos produtos da livraria depende de alguns fatores em determinadas épocas do ano, todos os produtos recebem a mesma porcentagem de acréscimo em relação ao preço de custo. Se o produto em questão for importado, é necessário considerar a cotação do dólar. Em algumas situações (promoções, por exemplo), é preciso atualizar todos os produtos de um determinado gênero. E, no caso específico dos livros didáticos, o cálculo do preço de venda é diferente dos demais.

Em outras linguagens de programação, como **não é possível termos duas funções** (blocos de código equivalentes a **métodos**) como o mesmo nome, nos depararíamos com a necessidade de criar funções nomeadas (calcularPrecoVenda1, calcularPrecoVenda2, calcularPrecoVenda3, calcularPrecoVenda4 ou calcularPrecoVendaNormal, calcularPrecoVendaImportado, calcularPrecoVendaPorGenero e

calcularPrecoVendaLivroDidatico e assim sucessivamente). Dessa forma, além de nomes extensos e muitas vezes estranhos, teríamos uma quantidade bem maior de nomes de funções para documentar no sistema.

Em Java, para situações desse tipo, usamos a sobrecarga que considera a identificação do método pela assinatura e não somente pelo nome. Como já vimos, a assinatura de um método é composta por:

- Nome do método e,
- Passagem de parâmetro.

Assim, é possível definir os métodos com o mesmo nome (calcularPrecoVenda, como no diagrama) e alternar a passagem de parâmetros. Observe então como ficaria os métodos na classe Produto.

```
public void CalcularPrecoVenda() {  
    // aplicando aumento no preco de 20%  
    this.setPrecoVenda(this.getPrecoCusto() * 1.2);  
}  
  
public void CalcularPrecoVenda(double Dolar) {  
    // aplicando aumento no preco pelo cotacao Dolar  
    this.setPrecoVenda(this.getPrecoCusto() * Dolar);  
}  
  
public void CalcularPrecoVenda(String genero) {  
    // aplicando aumento no preco dependente do genero  
    if (this.getGenero().equals(genero)) {  
        this.setPrecoVenda(this.getPrecoCusto() * 1.20);  
    }  
}
```

E na classe Livro ficaria assim:

```
public void CalcularPrecoVenda(boolean didatico) {  
    // aplicando aumento no preco se livro for didatico  
    if (didatico) {  
        this.setPrecoVenda(this.getPrecoCusto() * 1.10);  
    }  
}
```

Os pontos importantes na codificação anterior são:

- A diferenciação das assinaturas dos métodos se baseia na quantidade e no tipo de dado dos parâmetros.
- A sobrecarga pode ser realizada na mesma classe ou em subclasses, e os conceitos de herança são aplicados na utilização dos objetos. Em nosso exemplo, um objeto do tipo Livro tem quatro métodos `CalcularPrecoVenda()`, já na classe CD e DVD temos somente três.

No método principal (main), teremos o que mostra a figura abaixo quando digitarmos o (.) ponto no objeto Livro:

```
public static void main(String[] args) {  
    Livro livro = new Livro();  
  
    livro.Calcular  
        CalcularPrecoVenda(): void  
        CalcularPrecoVenda(boolean didatico): void  
        CalcularPrecoVenda(double Dolar): void  
        CalcularPrecoVenda(String genero): void  
}
```

O Java identificará o método que dever ser executado de acordo com a chamada realizada, como mostra o exemplo abaixo:

```
livro.CalcularPrecoVenda(1.9);
```

Como está sendo passado um valor do tipo double, será executado o método `CalcularPrecoVenda` que considera a cotação do dólar.

Exemplo-1: Faça um programa que receba o número de horas trabalhadas, o valor do salário-mínimo e o número de horas extras trabalhadas, calcule e mostre o salário a receber, seguindo as regras abaixo. Construa um método onde é possível passar o código do funcionário para cálculo específico.

- 1) A hora trabalhada equivale 1/48 do salário-mínimo
- 2) A hora extra vale 1/40 do salário-mínimo
- 3) O salário bruto equivale ao número de horas trabalhadas multiplicado pelo valor da hora trabalhada
- 4) A quantia a receber pelas horas extras equivale ao número de horas extras trabalhadas multiplicado pelo valor da hora extra
- 5) A salário a receber equivale ao salário bruto mais a quantia a receber pelas horas extras.

```
package sobrecarga;

import javax.swing.JOptionPane;

public class SobreCarga {

    public static void main(String[] args) {
        FolhaPagto folha;

        // instanciando a classe e passando os
        // parametros para o construtor da classe
        folha = new FolhaPagto(140, 1320, 80);

        JOptionPane.showMessageDialog(null, folha.CalculaFolha("74ABF23-J"));
    }
}
```



```
package sobrecarga;

public class FolhaPagto {
    // declarando os atributos da classe
    private double NHTrab;
    private double salMinimo;
    private double NHExtras;

    //Construtor da classe
    public FolhaPagto() {

    }

    // aplicando a sobrecarga
    public FolhaPagto(double NHTrab, double salMinimo, double NHExtras) {
        this.NHTrab = NHTrab;
        this.salMinimo = salMinimo;
        this.NHExtras = NHExtras;
    }

    // metodo da classe
    public String CalculaFolha() {
        String aux = "\nFolha de Pagto\n\n";

        double vHTrab = this.salMinimo / 48;
        double vHExtra = this.salMinimo / 40;
        double salBruto = (vHTrab * this.NHTrab);
        double quantiaHExtras = (vHExtra * this.NHExtras);
        double salReceber = (salBruto + quantiaHExtras);

        aux += "Hora Trabalhada R$ " + vHTrab + "\n";
        aux += "Hora Extra Trabalhada R$ " + vHExtra + "\n";
        aux += "Sal Bruto R$ " + salBruto + "\n";
        aux += "Quantia a receber HE R$ " + quantiaHExtras + "\n";
        aux += "Salario a receber R$ " + salReceber + "\n";

        return aux;
    }

    // aplicando a sobrecarga
    public String CalculaFolha(String codigoFunc) {
        String aux = "Funcionario não encontrado!";

        if (codigoFunc.equals("74ABF23-J")) {
            aux = "\nFolha de Pagto\n\n";
        }
    }
}
```



```
        double vHTrab = this.salMinimo / 48;
        double vHExtra = this.salMinimo / 40;
        double salBruto = (vHTrab * this.NHTrab);
        double quantiaHExtras = (vHExtra * this.NHExtras);
        double salReceber = (salBruto + quantiaHExtras);

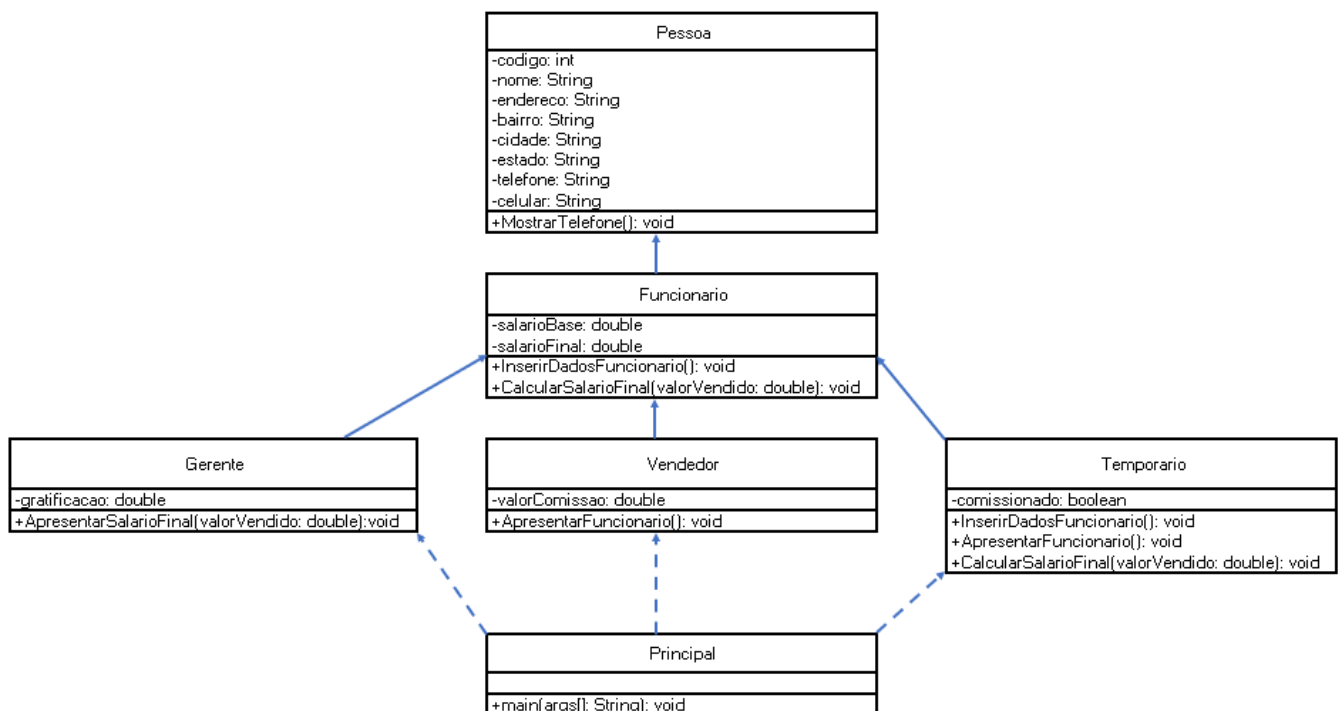
        aux += "Hora Trabalhada R$ "      + vHTrab      + "\n";
        aux += "Hora Extra Trabalhada R$ " + vHExtra    + "\n";
        aux += "Sal Bruto R$ "            + salBruto    + "\n";
        aux += "Quantia a receber HE R$ " + quantiaHExtras + "\n";
        aux += "Salario a receber R$ "    + salReceber  + "\n";
    }

    return aux;
}

// metodos get's e set's (modificadores)
public double getNHTrab() {
    return NHTrab;
}
public void setNHTrab(double NHTrab) {
    this.NHTrab = NHTrab;
}
public double getSalMinimo() {
    return salMinimo;
}
public void setSalMinimo(double salMinimo) {
    this.salMinimo = salMinimo;
}
public double getNHExtras() {
    return NHExtras;
}
public void setNHExtras(double NHExtras) {
    this.NHExtras = NHExtras;
}
}
```

Orientação a objetos - Sobrescrita

A sobre escrita de métodos está diretamente relacionada com a herança e consiste em reescrever um método herdado, mudando seu comportamento, mas mantendo exatamente a assinatura. Para exemplificar, utilizaremos o cálculo do salário final dos funcionários conforme diagrama abaixo:



Na superclasse Funcionário, foi implementado o método **CalcularSalarioFinal** considerando uma regra geral de cálculo (somar 10% do valor vendido ao salário base). Consequentemente, esse método foi herdado por suas subclasses **Gerente**, **Vendedor** e **Temporário**. O problema é que, de acordo com as normas da empresa, o cálculo do salário dos gerentes e dos temporários é diferente. Para os vendedores, o método **CalcularSalarioFinal** herdado está correto, porém, para **Gerente** e **Temporário**, não. O problema pode ser solucionado com a **sobrescrita** dos métodos **CalcularSalarioFinal** nas classes **Gerente** e **Temporário**. Na classe **Funcionário**, o método foi codificado da seguinte forma:

```

public void CalcularSalarioFinal(double valorVendido) {
    this.setSalarioFinal( this.getSalarioBase() + (valorVendido * 0.1) );
}
  
```

Já na subclasse Gerente fica da seguinte maneira:

```
public void CalcularSalarioFinal(double valorVendido) {  
    double meta = Double.parseDouble( JOptionPane.showInputDialog(  
        "Digite a meta de vendas do mês: "  
    ));  
  
    if (valorVendido > meta) {  
        this.setGratificacao(valorVendido * 0.15);  
    } else {  
        this.setGratificacao(0);  
    }  
    this.setSalarioFinal( this.getSalarioBase() + this.getGratificacao() );  
}
```

E na subclasse Temporário ficou da seguinte forma:

```
public void CalcularSalarioFinal(double valorVendido) {  
  
    if (this.comissionado) {  
        this.setSalarioFinal( this.getSalarioBase() + (valorVendido * 0.1) );  
    } else {  
        this.setSalarioFinal( this.getSalarioBase() );  
    }  
}
```

Os Objetos que representarão um gerente, um vendedor e um temporário serão instanciados a partir das classes Gerente, Vendedor e Temporário. Nesse momento (da criação dos objetos), o Java considera a estrutura de cada subclasse, a qual dá origem ao objeto. Então, um objeto do tipo Gerente considera a codificação do método CalcularSalarioFinal da classe Gerente; um do tipo Temporário leva em conta a codificação da classe Temporário, e um do tipo Vendedor focaliza o método herdado, mas todos são invocados exatamente da mesma forma, conforme mostra abaixo:

```
Gerente gerente = new Gerente();  
Vendedor vendedor = new Vendedor();  
Temporario temporario = new Temporario();  
  
gerente.CalcularSalarioFinal(5000);  
vendedor.CalcularSalarioFinal(5000);  
temporario.CalcularSalarioFinal(5000);
```

A sobrescrita de métodos também proporciona vantagens relacionadas ao gerenciamento polimórfico de objetos.

Exemplo-1: Todo professor de mestrado deve ser um professor de graduação. Os professores de graduação possuem os seguintes atributos: **matrícula, nome, salário bruto e quantidade de disciplinas**.

Além dos atributos de um professor de graduação, os professores de mestrado também possuem: **ano do término do doutorado, quantidade de artigos científicos escritos**.

O salário de um professor de graduação é calculado da seguinte forma:

$$= \text{salário bruto} - (\text{salário bruto} * 0.2) + \text{quantidade de disciplinas} * 50$$

O salário de um professor de mestrado é calculado da seguinte forma:

$$= \text{salário do professor de graduação} + \text{quantidade de artigos científicos} * 150$$

Na classe do método main, instancie um objeto do tipo professor de graduação, e outro do tipo professor de mestrado, e exiba o salário líquido de cada um.

```
package exemplosobrescrita1;

public class Graduacao {
    // declarando os atributos da classe
    public double salProfessor;
    public int matricula;
    public String nome;
    public double salBruto;

    // metodo da classe
    public void CalculaSalario() {
        this.salProfessor = this.salBruto - (this.salBruto * 0.2);
    }
}
```



```
package exemplosobrescrita1;

public class ProfessorGraduacao extends Graduacao {
    public double qtdeDisc;

    // metodo da classe sendo sobrescrita
    // quer dizer, manter a assinatura do metodo (nome e parametro)
    // reescrever o conteudo;
    @Override
    public void CalculaSalario() {

        super.salProfessor += this.qtdeDisc * 50;

    }
}
```

```
package exemplosobrescrita1;

public class ProfessorMestrado extends Graduacao {
    public String anoTermDoc;
    public double qtdeArtigos;

    // metodo da classe sendo sobrescrita
    // quer dizer, manter a assinatura do metodo (nome e parametro)
    // reescrever o conteudo;
    @Override
    public void CalculaSalario() {

        this.salProfessor += this.qtdeArtigos * 150;

    }
}
```

```
package exemplosobrescrita1;

import javax.swing.JOptionPane;

public class Principal {

    public static void main(String[] args) {
        ProfessorGraduacao profGrad = new ProfessorGraduacao();
        ProfessorMestrado profMes = new ProfessorMestrado();

        String aux;
        profGrad.matricula = 1257835;
        profGrad.nome = "Professor de graduação";
        profGrad.qtdeDisc = 20;
        profGrad.salBruto = 3200;
        profGrad.CalculaSalario();
        // preparando a impressao para o prof. de graduacao
        aux = "\nDados Professor graduação\n";
        aux += "Matricula: " + profGrad.matricula + "\n";
        aux += "Nome prof: " + profGrad.nome + "\n";
        aux += "Qtd Disc.: " + profGrad.qtdeDisc + "\n";
        aux += "Sal. Prof: R$ " + profGrad.salProfessor + "\n";

        profMes.matricula = 98765;
        profMes.nome = "Professor de Mestrado";
        profMes.anoTermDoc = "2 anos";
        profMes.qtdeArtigos = 15;
        profMes.salBruto = 4200;
        profMes.CalculaSalario();
        // preparando a impressao para o prof. de graduacao
        aux += "\n\nDados Professor mestrado\n";
        aux += "Matricula: " + profMes.matricula + "\n";
        aux += "Nome prof: " + profMes.nome + "\n";
        aux += "Qtd Artigos: " + profMes.qtdeArtigos + "\n";
        aux += "Term. Docto: " + profMes.anoTermDoc + "\n";
        aux += "Sal. Prof: R$ " + profMes.salProfessor + "\n";

        // impressao
        JOptionPane.showMessageDialog(null, aux);
    }
}
```

Exemplo-2: Desenvolva um projeto em Java que leia um número inteiro e imprima se o nro escolhido aleatoriamente pela classe Random é um nro primo.

Desenvolva um método em uma subclasse praticando a sobrescrita:

- Antecessor primo e seu sucessor também um nro primo.

```
package exemplosobrescrita2;

public class NroPrimos {

    // metodo da classe
    public String CalculaSePrimo(int nro) {
        int i, cont=0;

        for (i = 1; i <= nro; i++) {
            if (nro % i == 0) {
                cont++;
            }
        }
        if (cont <= 2) {
            return "O nro: " + nro + " é primo!";
        } else {
            return "O nro: " + nro + " não é primo!";
        }
    }
}
```

```
package exemplosobrescrita2;

public class CalculaPrimoAntecessor extends NroPrimos {

    // praticando a sobrescrita
    @Override
    public String CalculaSePrimo(int nro) {
        int i, cont;

        while (true) {
            cont = 0;
            for (i = 1; i <= nro; i++) {
                if (nro % i == 0) {
                    cont++;
                }
            }
            if (cont <= 2) {
                return "O nro: " + nro + " antecessor é primo!";
            } else {
                nro -= 1;
            }
        }
    }
}
```

```
package exemplosobrescrita2;

public class CalculaPrimoSucessor extends NroPrimos {

    // praticando a sobrescrita
    @Override
    public String CalculaSePrimo(int nro) {
        int i, cont;

        while (true) {
            cont = 0;
            for (i = 1; i <= nro; i++) {
                if (nro % i == 0) {
                    cont++;
                }
            }
            if (cont <= 2) {
                return "O nro: " + nro + " sucessor é primo!";
            } else {
                nro += 1;
            }
        }
    }
}
```

```
/*
    Desenvolva um projeto em Java que leia um número inteiro e imprima se
    o nro digitado pelo usuario é um nro primo.

    Desenvolva um metodo em uma subclasse praticando a sobrescrita:
    - antecessor primo e seu sucessor também um nro primo.
*/
package exemplosobrescrita2;

import java.util.Random;
import javax.swing.JOptionPane;

public class Principal {

    public static void main(String[] args) {
        CalculaPrimoAntecessor calcA = new CalculaPrimoAntecessor();
        CalculaPrimoSucessor calcS = new CalculaPrimoSucessor();
        Random rand = new Random();

        int nro = rand.nextInt(1000);
        String aux = "";

        aux = "Nro sorteado: " + nro + "\n";
        aux += calcA.CalculaSePrimo(nro) + "\n";
        aux += calcS.CalculaSePrimo(nro) + "\n";
        JOptionPane.showMessageDialog(null, aux);

    }

}
```

Orientação a objetos – Polimorfismo

O polimorfismo é a possibilidade de utilizar um objeto “**como se fosse**” um outro. Embora o conceito seja esse, algumas publicações relacionadas ao JAVA e à orientação de objetos fazem abordagens diferentes. Então, podemos considerar basicamente **três tipos de polimorfismo**:

- Métodos;
- Classe;
- Interface.

1. De métodos

Muitos autores consideram **polimorfismo a possibilidade de utilizar dois ou mais métodos com a mesma assinatura**, mas com comportamentos (**codificação**) diferentes. Basta lembrar que já abordamos um recurso da linguagem Java que permite exatamente isso: **a sobrescrita**.

A questão não é avaliar se essa visão está correta ou não. Mesmo porque, de certa forma, **a sobrescrita possibilita o uso de um método que pode assumir várias formas** (**executar tarefas diferentes, de acordo com sua codificação**) a partir da mesma chamada, sendo diferenciado pela classe que deu origem ao objeto, Isso justificaria chamar esse recurso de polimorfismo, mas acreditamos que é melhor definido como sobrescrita.

2. De Classe

Considerando uma hierarquia de classes, temos, em uma superclasse, a generalização de um tipo e, em suas subclasses, a especialização do mesmo tipo, imagine a seguinte situação:

- Se colocarmos uma cesta à disposição dos clientes da livraria e nela estiver escrito “Coloque aqui seus produtos”, estes “produtos” podem ser livros, cd’s ou dvd’s. Ou ainda, todos eles juntos.

Outro exemplo:

- Na livraria, existe uma porta de acesso ao estoque e nela há uma placa com o aviso “Entrada permitida somente para funcionários”

Tanto pode entrar um vendedor como um gerente, porque ambos são funcionários.

Esse mesmo princípio se aplica aos programas em Java. Se definirmos um **método que recebe um objeto do tipo Produto por parâmetro**, podemos passar qualquer objeto instanciado a partir de uma subclasse de Produto que ele será aceito. Da mesma forma, **se um método espera um objeto do tipo Funcionário**, é possível passar um objeto instanciado a partir da classe Vendedor, por exemplo, que será aceito sem problemas.

O raciocínio é o seguinte: “**Um vendedor é um funcionário**”, assim como, “**Um livro é um produto**”. A diferença é que vendedor foi definido a partir de uma especialização da classe Funcionário. Assim, pode ter atributos e métodos específicos referentes a vendedores, porém, não deixa de ser um Funcionário.

Apesar de um livro poder ser utilizado “**como se fosse**” um Produto, não deixa de “**ser**” um livro e de ter as características específicas de livro. **O polimorfismo, portanto, é a possibilidade de utilizar um objeto como se fosse outro e não o transformar em outro objeto**. O uso do polimorfismo de classe (ou de tipo) é mostrado na figura abaixo, que mostra o processo de venda da livraria.

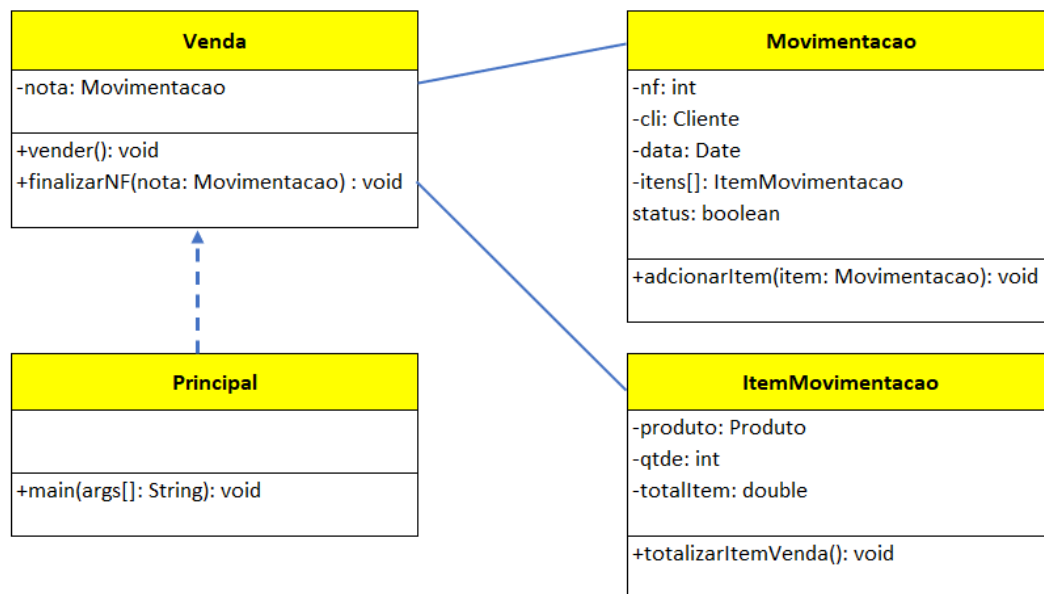
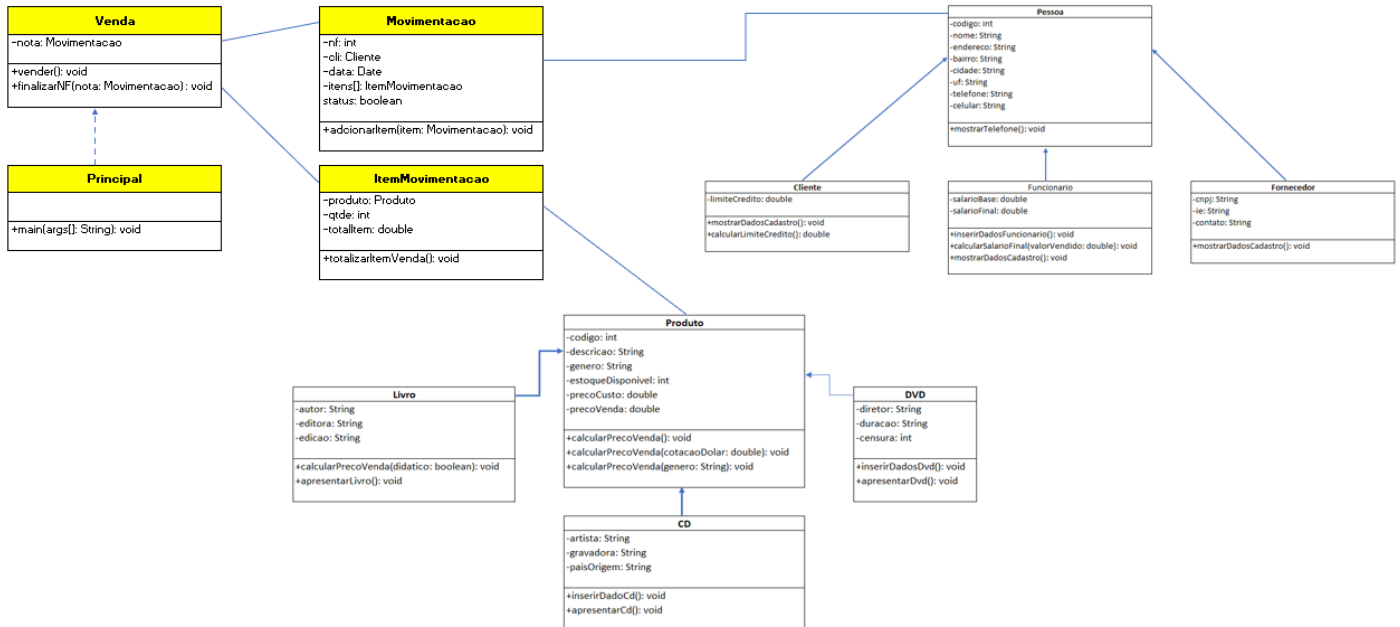


Diagrama completo do sistema da livraria



Resumo: Polimorfismo é a capacidade de uma referência de um tipo genérico referenciar um objeto de um tipo mais específico.

Exemplo 1:

Veja que o tipo da **variável é Carro**, mas o objeto colocado nela não é Carro e sim **Ferrari e Fusca**.

```

package com.mycompany.polimorfismoexemplo1;

public interface Carro {

    public void acelerar();

}

package com.mycompany.polimorfismoexemplo1;

public class Ferrari implements Carro {

    @Override
    public void acelerar() {
        System.out.println(x: "Ferrari acelerando...");
    }

}

package com.mycompany.polimorfismoexemplo1;

public class Fusca implements Carro {

    @Override
    public void acelerar() {
        System.out.println(x: "Fusca tentando acelerar...");
    }

}
    
```

```

package com.mycompany.polimorfismoexemplo1;

public class PolimorfismoExemplo1 {

    public static void main(String[] args) {

        Carro c = new Ferrari();
        c.acelerar();

        c = new Fusca();
        c.acelerar();

    }

}
    
```

Exemplo 2:

Novamente, o tipo da **variável é Conexão**, mas não referência um objeto do tipo Conexão, mas sim um objeto **DialUp** ou **Adsl**, essa variável com pode referenciar qualquer objeto que implemente a interface **Conexão** ou estenda a classe, se for uma.

```
package com.mycompany.polimorfismoexemplo2;

public interface Conexao {

    public void conectar();

}
```

```
package com.mycompany.polimorfismoexemplo2;

public class DialUp implements Conexao {

    @Override
    public void conectar() {
        System.out.println(x: "Modem discando...");
    }

}
```

```
package com.mycompany.polimorfismoexemplo2;

public class Adsl implements Conexao {

    @Override
    public void conectar() {
        System.out.println(x: "Adsl conectado...");
    }

}
```

```
package com.mycompany.polimorfismoexemplo2;

public class PolimorfismoExemplo2 {

    public static void main(String[] args) {
        Conexao con = new DialUp();
        con.conectar();

        con = new Adsl();
        con.conectar();
    }

}
```

Exemplo 3:

Vamos para a nossa loja de carros, onde você é o programador Java de lá.

Lembra que aprendeu, através de exemplos, a criar uma classe bem genérica "interface" chamada "Carro" e depois criamos várias subclasses, de fuscas, ferraris etc. Imagine que, todo ano, todos na empresa tem um aumento, a Ferrari teve aumento de 5%. o fusca terá aumento de 3% e o gol terá de 1%.

Note que, embora todos sejam "Carro", cada objeto terá que calcular seu aumento de forma diferente, pois terão diferentes valores de aumento.

Como criar, então, um método na superclasse que atenda todas essas necessidades diferentes?

Não é na superclasse que se resolve, mas nas subclasses, criando o método "AumentoAnual()" em cada uma. Ou seja, vai criar vários métodos, e para fazer o aumento realmente ocorrer de maneira correta, é só invocar o método do objeto específico.

Portanto: `objetoFerrari.aumento()` é diferente de `objetoFusca.aumento()`.

Note que **usamos o mesmo nome do método para todas as subclasses**, porém **cada método é diferente um do outro**. Isso é o **polimorfismo em ação**: embora todos os objetos sejam "Carro", eles terão uma forma diferente de agir, pois implementamos os métodos de maneira diferente. Apenas invocamos, e todo objeto sabe exatamente o que fazer.

Por isso o nome polimorfismo, pois cada objeto terá sua forma própria de como rodar, pois os métodos 'AumentoAnual()' dos objetos são diferentes.

```
package com.mycompany.concessionaria;

public interface Carro {

    public double AumentoAnual();

    public void ApresentaCarro();

}

package com.mycompany.concessionaria;

public class Fusca implements Carro {
    public double valorCarro;

    public Fusca() {
        this.valorCarro = 50000;
    }

    @Override
    public double AumentoAnual() {
        this.valorCarro *= 1.10;
        return this.valorCarro;
    }

    @Override
    public void ApresentaCarro() {
        System.out.println("Valor do Fusca: R$ " + String.format("%.2f", this.valorCarro));
    }

}
```

```
package com.mycompany.concessionaria;

public class Golf implements Carro {
    public double valorCarro;

    public Golf() {
        this.valorCarro = 180000;
    }

    @Override
    public double AumentoAnual() {
        this.valorCarro *= 1.20;
        return this.valorCarro;
    }

    @Override
    public void ApresentaCarro() {
        System.out.println("Valor do Golf: R$ " + String.format("%.2f", this.valorCarro));
    }
}

package com.mycompany.concessionaria;

public class Renault implements Carro {
    public double valorCarro;

    public Renault() {
        this.valorCarro = 100000;
    }

    @Override
    public double AumentoAnual() {
        this.valorCarro *= 1.07;
        return this.valorCarro;
    }

    @Override
    public void ApresentaCarro() {
        System.out.println("Valor do Renault: R$ " + String.format("%.2f", this.valorCarro));
    }
}

package com.mycompany.concessionaria;

public class Concessionaria {

    public static void main(String[] args) {
        Carro objeto;

        objeto = new Fusca();
        objeto.AumentoAnual();
        objeto.ApresentaCarro();

        objeto = new Golf();
        objeto.AumentoAnual();
        objeto.ApresentaCarro();
    }
}
```

```
objeto = new Renault();  
objeto.AumentoAnual();  
objeto.ApresentaCarro();  
  
}  
}
```

Outra vantagem do polimorfismo: você já viu que, criando o método “AumentoAnual()” em toda as subclasses, ela agirão de maneira independente da superclasse e diferente de outros objetos. Agora, quando chegar outro carro na sua loja você de adicionar o método “AumentoAnual()”, e terá um novo tipo de objeto, sem grandes alterações no código.