

# **Programação Orientada a Objetos**

## **Notas de Aula**

Prof. Marcelo de Souza



Universidade do Estado de Santa Catarina  
Centro de Educação Superior do Alto Vale do Itajaí

Versão compilada em 2 de julho de 2018.

Este material é utilizado em parte das aulas da disciplina de Programação I (25PRO1) do curso de Bacharelado em Engenharia de Software da Universidade do Estado de Santa Catarina (UDESC Ibirama).

**Contato:** `marcelo.desouza@udesc.br`

Esta obra está licenciada sob uma [Licença Creative Commons](#) (Atribuição-SemDerivações-SemDerivados 3.0 Brasil).

# Sumário

<b>I. Conceitos básicos</b>	<b>7</b>
<b>1. Linguagem Java</b>	<b>9</b>
1.1. Variáveis e constantes . . . . .	9
1.2. Tipos primitivos em Java . . . . .	9
1.3. Leitura de valores do usuário . . . . .	10
1.4. Operadores aritméticos . . . . .	11
1.5. Operadores lógicos . . . . .	11
1.6. Operadores relacionais . . . . .	13
1.7. Estruturas condicionais . . . . .	13
1.7.1. Estruturas condicionais simples . . . . .	13
1.7.2. Estruturas condicionais aninhadas . . . . .	14
1.7.3. Estruturas condicionais e múltipla escolha . . . . .	15
1.8. Laços de repetição . . . . .	15
1.8.1. Laço de repetição contado – “for” . . . . .	15
1.8.2. Laço de repetição condicional com teste no início – “while” . . . . .	16
1.8.3. Laço de repetição condicional com teste no final – “do” . . . . .	17
1.9. Métodos . . . . .	17
1.9.1. Métodos sem retorno e sem parâmetros . . . . .	18
1.9.2. Métodos com retorno e sem parâmetros . . . . .	18
1.9.3. Métodos com parâmetros . . . . .	18
1.10. Vetores . . . . .	19
1.10.1. Vetores unidimensionais . . . . .	19
1.10.2. Vetores multidimensionais . . . . .	20
1.11. Manipulação de String . . . . .	21
<b>2. Introdução à orientação a objetos</b>	<b>23</b>
2.1. Visão geral . . . . .	23
2.2. Abstração . . . . .	23
2.3. Classes . . . . .	24
2.4. Atributos e Métodos . . . . .	25
2.5. Objetos . . . . .	25
2.6. Implementação de classes . . . . .	26
2.7. Encapsulamento . . . . .	27
2.8. Métodos acessores . . . . .	28
2.9. Métodos construtores . . . . .	29
2.10. Métodos destrutores . . . . .	30
2.11. Sobrecarga de operações . . . . .	31
2.12. Exemplo – Compra . . . . .	31

<b>II. Relacionamentos entre classes</b>	<b>35</b>
<b>3. Visão geral</b>	<b>37</b>
3.1. Tipos de relacionamentos . . . . .	37
3.2. Representação UML . . . . .	37
<b>4. Associações simples</b>	<b>39</b>
4.1. Associação com multiplicidade um (1) . . . . .	39
4.1.1. Caso unidirecional . . . . .	39
4.1.2. Caso bidirecional . . . . .	43
4.2. Exemplo – Gerente e Sala . . . . .	44
4.3. Associação com multiplicidade muitos (*) . . . . .	49
4.3.1. Caso unidirecional . . . . .	49
4.3.2. Caso bidirecional . . . . .	53
4.4. Exemplo – Clube e Sócio . . . . .	55
<b>5. Agregação e composição</b>	<b>61</b>
5.1. Relacionamento todo-parte . . . . .	61
5.2. Agregação . . . . .	62
5.2.1. Caso com multiplicidade 1 . . . . .	63
5.2.2. Caso com multiplicidade muitos . . . . .	64
5.3. Composição . . . . .	65
5.3.1. Caso com multiplicidade 1 . . . . .	66
5.3.2. Caso com multiplicidade muitos . . . . .	67
<b>6. Dependência</b>	<b>69</b>
6.1. Dependência no argumento de um método . . . . .	69
6.2. Dependência no retorno de um método . . . . .	71
6.3. Dependência por uso interno . . . . .	73
<b>7. Herança e polimorfismo</b>	<b>77</b>
7.1. Herança . . . . .	77
7.2. Sobrescrita de método . . . . .	80
7.3. Polimorfismo . . . . .	83
7.4. Exemplo – Veículos . . . . .	84
7.4.1. Estrutura das classes . . . . .	84
7.4.2. Implementação das entidades . . . . .	85
7.4.3. Uso das classes . . . . .	87
<b>8. Classes abstratas</b>	<b>91</b>
8.1. Conceitos e benefícios . . . . .	91
8.2. Implementação . . . . .	92
8.3. Métodos abstratos . . . . .	93
<b>9. Realização</b>	<b>95</b>
9.1. Interfaces . . . . .	95
9.2. Mais polimorfismo . . . . .	97

<b>III. Tópicos adicionais</b>	<b>99</b>
<b>10. Interfaces gráficas</b>	<b>101</b>
10.1. Construção usando o NetBeans . . . . .	101
10.2. Barra de menu, eventos e ações . . . . .	103
10.3. Entidade Veiculo . . . . .	106
10.4. Componentes da tela . . . . .	107
10.5. Funcionalidades . . . . .	107
10.5.1. Salvar registro . . . . .	108
10.5.2. Cancelar preenchimento . . . . .	109
10.5.3. Excluir registro . . . . .	110
10.5.4. Pesquisar registro . . . . .	110
10.5.5. Excluir registro . . . . .	111
10.6. Outros componentes . . . . .	112
<b>11. Persistência em arquivos</b>	<b>113</b>
11.1. Operação de escrita . . . . .	113
11.2. Operação de leitura . . . . .	114
11.3. Exemplo – Cadastro de Pessoas . . . . .	114
11.3.1. Entidade Pessoa . . . . .	114
11.3.2. Classe para persistência de dados . . . . .	115
11.3.3. Escrita de uma pessoa . . . . .	116
11.3.4. Leitura de uma pessoa . . . . .	116
11.3.5. Leitura de todas as pessoas . . . . .	117
11.3.6. Exclusão de uma pessoa . . . . .	118
<b>IV. Apêndices</b>	<b>119</b>
<b>A. Lista de exercícios</b>	<b>121</b>
1. Revisão sobre Java . . . . .	121
2. Conceitos básicos de orientação a objetos . . . . .	124
3. Associações simples . . . . .	126
4. Agregação e composição . . . . .	129
5. Dependência . . . . .	132
6. Herança e polimorfismo . . . . .	134
<b>Referências Bibliográficas</b>	<b>135</b>



# **Parte I.**

## **Conceitos básicos**





# 1. Linguagem Java

## 1.1. Variáveis e constantes

Uma variável é um mapeamento para um espaço alocado de memória, no qual se pode armazenar valores de um determinado tipo. O exemplo abaixo mostra a declaração de variáveis, atribuição de valores e sua recuperação.

```
1  int idade;           //Declaração
2  idade = 10;          //Atribuição
3  int valor = idade;    //Recuperação
4  System.out.println(idade); //Recuperação e apresentação
```

Uma constante nada mais é do que uma variável que possui um valor predeterminado, com alteração não permitida. O trecho de código abaixo exemplifica a declaração de uma constante (com seu valor fixo) e a recuperação deste valor. A criação de uma constante é feita pelo uso da palavra reservada **final**, que torna seu valor fixo e não permite alteração (valor final).

```
1  final int matricula = 512010681; //Declaração e atribuição
2  int aluno = matricula;           //Recuperação
3  System.out.println(matricula);   //Recuperação
```

## 1.2. Tipos primitivos em Java

Existem quatro categorias básicas para variáveis em Java: numérico (armazenamento de números), caracter (armazenamento de caracteres), alfanumérico (armazenamento de valores textuais) e lógico (armazenamento de valores lógicos – verdadeiro e falso). A Tabela 1.1 resume os tipos primitivos em Java para cada categoria e o tamanho ocupado em memória por variáveis de cada tipo.

O trecho de código abaixo mostra o uso de variáveis dos diferentes tipos, com o armazenamento de valores e a recuperação para apresentação em tela.

```
1  String nome = "João";
2  int idade = 15;
3  char sexo = 'M';
4  boolean aprovado = true;
5  System.out.println("Dados do acadêmico")
```

Categoria	Tipo	Tamanho
numérica	byte	8 bits
numérica	short	16 bits
numérica	int	32 bits
numérica	long	64 bits
numérica	float	32 bits
numérica	double	64 bits
alfanumérica	char	16 bits
lógica	booleana	1 bit

Tabela 1.1.: Tipos primitivos em Java

```

6      + "\nNome: " + nome
7      + "\nIdade: " + idade
8      + "\nSexo: " + sexo
9      + "\nAprovado: " + aprovado);

```

### 1.3. Leitura de valores do usuário

A classe `Scanner` pode ser utilizada para a leitura de valores a partir do console. Esta classe memoriza o valor digitado pelo usuário e armazena em uma variável indicada. O exemplo abaixo mostra a criação de um objeto (`scanner`) desta classe e a posterior leitura de valores.

```

1  Scanner scanner = new Scanner(System.in);
2  int valorInteiro = scanner.nextInt();
3  String valorTextual = scanner.next();
4  double d = scanner.nextDouble();

```

Uma alternativa ao uso do console como entrada e saída de dados consiste em apresentar ao usuário uma caixa de diálogo com as informações de saída ou com a solicitação de uma entrada. A classe `JOptionPane` fornece os métodos necessários a esta tarefa. O método `showMessageDialog` permite a apresentação de uma mensagem ao usuário, enquanto o método `showInputDialog` permite solicitar um valor ao usuário. Note que o método retorna um valor do tipo `String`, que deve ser convertido ao tipo desejado, caso diferente. O exemplo abaixo mostra o uso da classe `JOptionPane` para as tarefas supracitadas.

```

1  JOptionPane.showMessageDialog(null, "Mensagem ao usuário");
2  int valorInteiro =
    ↳ Integer.parseInt(JOptionPane.showInputDialog("Digite um valor
    ↳ inteiro:"));
3  String valorTextual = JOptionPane.showInputDialog("Digite uma
    ↳ String:");
4  double d = Double.parseDouble(JOptionPane.showInputDialog("Digite um
    ↳ valor double: "));

```

## 1.4. Operadores aritméticos

Operadores aritméticos são funções e operadores predefinidos para realização de cálculos matemáticos. Eles atuam sobre valores ou variáveis numéricas. A Tabela 1.2 apresenta os principais operadores aritméticos disponíveis na linguagem Java, seus exemplos e sua descrição.

Operador	Exemplo	Descrição
=	<code>x = y</code>	O conteúdo da variável <code>y</code> é atribuído à variável <code>x</code> .
+	<code>x + y</code>	Soma o conteúdo de <code>x</code> e de <code>y</code> .
-	<code>x - y</code>	Subtrai o conteúdo de <code>y</code> do conteúdo de <code>x</code> .
*	<code>x * y</code>	Multiplicação conteúdo de <code>x</code> com o conteúdo de <code>y</code> .
/	<code>x / y</code>	Divide o conteúdo de <code>x</code> pelo conteúdo de <code>y</code> .
%	<code>x % y</code>	Obtém o resto da divisão inteira de <code>x</code> por <code>y</code> .
++	<code>x++</code>	Equivale a <code>x = x + 1</code> .
--	<code>x--</code>	Equivale a <code>x = x - 1</code> .

Tabela 1.2.: Operadores Aritméticos

O trecho de código abaixo exemplifica a aplicação dos operadores supracitados. Em expressões mais complexas, os operadores têm a mesma precedência que na matemática básica, a qual pode ser alterada com o uso de parêntesis.

```
1  int a, b, resultado;
2  a = 10;
3  b = 3;
4  resultado = a + b;           //resultado -> 13
5  resultado = a - b;           //resultado -> 7
6  resultado = a * b;           //resultado -> 30
7  resultado = a / b;           //resultado -> 3
8  resultado = (a / b) + a ;    //resultado -> 13
9  resultado = a % b;           //resultado -> 1
10 resultado = a++;             //resultado -> ???
11 resultado = b--;             //resultado -> ???
```

O resultado das expressões apresentadas nas linhas 10 e 11 é **10** e **3**, respectivamente. Isto se deve ao fato de que as operações `a++` e `b--` são realizadas após a atribuição, ou seja, após os valores originais serem armazenados na variável `resultado`. Para que as operações sobre as variáveis `a` e `b` sejam feitas antes da atribuição, deve-se utilizar `++a` e `--b`.

## 1.5. Operadores lógicos

Operadores lógicos são funções ou operadores predefinidos para atuação sobre valores ou variáveis booleanas (lógicas), avaliando seu conteúdo. Estes operadores retornam um

valor booleano (**true** ou **false**) como resultado. A Tabela 1.3 apresenta uma visão geral dos operadores lógicos (AND, OR e NOT). As Tabelas 1.4, 1.5 e 1.6 detalham o funcionamento de cada operador, apresentando a tabela verdade de cada um.

Operador	Exemplo	Descrição
E/AND – &&	x && y	Representa o conteúdo de x <b>E</b> o conteúdo de y
OU/OR –	x    y	Representa o conteúdo de x <b>OU</b> o conteúdo de y
NÃO/NOT – !	!x	Representa o valor inverso de x (não x).

Tabela 1.3.: Visão geral dos operadores lógicos

Valor de x	Operador	Valor de y	Resultado
TRUE	&&	TRUE	TRUE
TRUE	&&	FALSE	FALSE
FALSE	&&	TRUE	FALSE
FALSE	&&	FALSE	FALSE

Tabela 1.4.: Tabela verdade para o operador AND

Valor de x	Operador	Valor de y	Resultado
TRUE		TRUE	TRUE
TRUE		FALSE	TRUE
FALSE		TRUE	TRUE
FALSE		FALSE	FALSE

Tabela 1.5.: Tabela verdade para o operador OR

Operador	Valor de x	Resultado
!	TRUE	FALSE
!	FALSE	TRUE

Tabela 1.6.: Tabela verdade para o operador NOT

O trecho de código a seguir mostra a aplicação dos operadores lógicos sobre variáveis booleanas.

```

1  boolean a, b, resultado;
2  a = true;
3  b = false;
4  resultado = a && b;    //resultado -> false
5  resultado = a || b;    //resultado -> true
6  resultado = !a;        //resultado -> false

```

## 1.6. Operadores relacionais

Operadores relacionais são funções ou operadores predefinidos para atuação sobre valores ou variáveis de diferentes tipos, comumente numéricos. Estes operadores avaliam o conteúdo de duas variáveis ou grandezas e os compara, retornando um valor lógico (**true** ou **false**). Em outras palavras, estes operadores relacionam duas variáveis ou duas grandezas, comparando-as.

Operador	Exemplo	Descrição
Igual – ==	<code>x==y</code>	Conteúdo de <code>x</code> é igual ao conteúdo de <code>y</code> ?
Diferente – !=	<code>x!=y</code>	Conteúdo de <code>x</code> é diferente do conteúdo de <code>y</code> ?
Maior – >	<code>x &gt; y</code>	Conteúdo de <code>x</code> é maior que o conteúdo de <code>y</code> ?
Menor – <	<code>x &lt; y</code>	Conteúdo de <code>x</code> é menor que o conteúdo de <code>y</code> ?
Maior ou igual – >=	<code>x &gt;= y</code>	Conteúdo de <code>x</code> é maior ou igual ao conteúdo de <code>y</code> ?
Menor ou igual – <=	<code>x &lt;= y</code>	Conteúdo de <code>x</code> é menor ou igual ao conteúdo de <code>y</code> ?

Tabela 1.7.: Operadores relacionais

O trecho de código a seguir mostra a aplicação dos diferentes operadores relacionais, avaliando o conteúdo das variáveis numéricas `a` e `b`.

```
1  int a, b;
2  boolean resultado;
3  a = 5;
4  b = 2;
5  resultado = (a == b); //resultado -> false
6  resultado = (a != b); //resultado -> true
7  resultado = (a > b); //resultado -> true
8  resultado = (a < b); //resultado -> false
9  resultado = (a >= b); //resultado -> true
10 resultado = (a <= b); //resultado -> false
```

## 1.7. Estruturas condicionais

Estruturas condicionais (ou estruturas de seleção) permitem definir um trecho de código que é executado apenas em condições determinadas. Ou seja, elas permitem que o fluxo de execução seja desviado, de acordo com condições predefinidas.

### 1.7.1. Estruturas condicionais simples

A instrução `if` permite criar uma estrutura condicional simples, que faz a verificação de uma condição e executa um trecho de código somente se a condição for verdadeira (retornar **true**). O trecho de código abaixo apresenta um exemplo da aplicação de uma estrutura condicional simples.

```
1  int a = 10;
2  int b = 5;
3  if(a > b) {
4      System.out.println("Este código só é executado se a > b");
5  }
```

A condição definida pela expressão `if` consiste em avaliar o conteúdo das variáveis `a` e `b` com o operador relacional `>`. Caso a expressão `a > b` for verdadeira (como no caso do exemplo), as instruções internas à estrutura condicional serão executadas.

É possível utilizar a cláusula `else` para definir um conjunto de instruções executadas exclusivamente quando a condição não for satisfeita (retornar `false`). O trecho de código a seguir exemplifica o uso da cláusula `else`.

```
1  int a = 10;
2  int b = 5;
3  if(a > b) {
4      System.out.println("Este código só é executado se a > b");
5  } else {
6      System.out.println("Este código só é executado se a <= b");
7  }
```

### 1.7.2. Estruturas condicionais aninhadas

É possível definir estruturas condicionais internas a outras estruturas condicionais. Dessa forma, é possível encadear e testar diferentes condições e sub-condições. O trecho de código abaixo mostra um exemplo de estruturas condicionais aninhadas.

```
1  int a = 10;
2  int b = 5;
3  if(a > b) {
4      if(b < 50) {
5          System.out.println("Este código só é executado se a > b e b <
↪ 50");
6      }
7  } else {
8      if(a > 30) {
9          System.out.println("Este código é executado quando a condição
↪ (a > b) não é satisfeita e a > 30");
10     }
11 }
```

No exemplo apresentado, uma primeira condição é testada (`a > b`). Caso ela seja verdadeira, a condição `b < 50` é então testada. Caso contrário, a condição `a > 30` é testada.

### 1.7.3. Estruturas condicionais e múltipla escolha

O Java (assim como outras linguagens) fornece um terceiro tipo de estrutura condicional para múltiplas escolhas. Estas estruturas testam o valor de uma variável e selecionam uma entre diferentes opções, executando o código correspondente. Um exemplo é apresentado no trecho de código abaixo. Atente para o uso da cláusula `break` em cada condição, evitando que condições não desejadas sejam satisfeitas.

```
1  int a = 2;
2
3  switch(a) {
4      case 1:
5          System.out.println("O valor de a é 1");
6          break;
7      case 2:
8          System.out.println("O valor de a é 2");
9          break;
10     case 3:
11         System.out.println("O valor de a é 3");
12         break;
13     default:
14         System.out.println("O valor de a não é nenhum dos valores
15     ↪     testados");
16         break;
17 }
```

## 1.8. Laços de repetição

Laços de repetição permitem a execução repetida de um trecho de código. Ou seja, o programador define a forma como um processo é realizado e o replica repetidas vezes. Exemplo: fazer a leitura das notas dos alunos da disciplina para cálculo da média. Como existem vários alunos, o processo de leitura e cômputo da média deve ser executado múltiplas vezes.

### 1.8.1. Laço de repetição contado – “for”

A primeira forma de implementação de laço de repetição é determinando o número de iterações do mesmo. Isso é feito através do laço `for`, no qual se define uma variável contadora, seu valor inicial, sua condição de parada e o incremento (ou decremento) a cada iteração. O trecho de código abaixo exemplifica o uso do laço de repetição `for`.

```
1  for(int i = 0; i < 10; i++) {
2      System.out.println("Este texto será impresso 10 vezes!");
3  }
```

O trecho de código a seguir mostra a aplicação do laço **for**. O primeiro laço (linhas 1 a 3) inicia em **10** e decrementa a variável contadora enquanto **i > 0**. Logo, o texto é impresso 10 vezes. O segundo laço (linhas 5 a 10) utiliza variáveis externas (**inicio** e **fim**) para controlar as iterações. A cláusula **break** permite quebrar a execução do laço, isto é, parar sua execução e pular para a linha subsequente a ele.

```
1  for(int i = 10; i > 0; i--) {  
2      System.out.println("Esse texto será impresso 10 vezes!");  
3  }  
4  
5  int inicio = 1, fim = 10;  
6  for(int i = inicio; i <= fim; i++) {  
7      System.out.println("Executando...");  
8      if(i > 100)  
9          break;  
10 }
```

### 1.8.2. Laço de repetição condicional com teste no início – “while”

Os laços de repetição condicionais repetem a execução do conjunto de instruções de acordo com uma condição. Ou seja, enquanto a condição for verdadeira, o código correspondente é executado. A primeira forma de laço de repetição condicional é o **while**, que define o teste condicional no início do laço. Um exemplo básico é apresentado a seguir, onde o conjunto de instruções é executado enquanto a condição **i < 10** for verdadeira. Atente para a necessidade de atualização da variável de controle (**i** – linha 4).

```
1  int i = 0;  
2  while (i < 10) {  
3      System.out.println("Este texto será impresso 10 vezes");  
4      i++;  
5  }
```

No exemplo a seguir, utilizamos uma variável booleana (**continua**) para controlar as repetições do laço. Igualmente, é necessário definir o caso em que esta variável torna-se **false**, evitando um laço infinito. No exemplo, a variável recebe **false** quando a variável **i** for maior que 1000 (linhas 4 a 7).

```
1  boolean continua = true;  
2  while(continua) {  
3      i *= 2;  
4      if(i > 1000) {  
5          System.out.println("i -> " + i);  
6          continua = false;  
7      }  
8  }
```



### 1.8.3. Laço de repetição condicional com teste no final – “do”

Analogamente ao método anterior, a estrutura `do..while` é um laço de repetição condicional, cujo teste é executado ao final da sua definição. Com isso, garante-se que o conjunto de instruções do laço será executado no mínimo uma vez, mesmo quando o condicional é falso. Isso se deve ao fato da estrutura primeiro executar as instruções e depois testar o condicional.

O código abaixo mostra o uso do laço `do..while`. A primeira definição do laço repete por 10 vezes, enquanto a condição `i < 10` se apresenta verdadeira. A segunda definição do laço possui uma condição falsa por definição (valor atribuído na linha 7). No entanto, a instrução (linha 10) é executada uma vez, pois o teste ocorre no final da estrutura de repetição.

```
1  int i = 0;
2  do {
3      System.out.println("Este texto será impresso 10 vezes");
4      i++;
5  } while (i < 10);
6
7  boolean continua = false;
8
9  do {
10     System.out.println("Esse texto será exibido uma vez, pois o teste
    ↳ é realizado no final!");
11 } while(continua);
```

## 1.9. Métodos

O uso de métodos em programação segue o conceito de dividir para conquistar. Ou seja, um problema pode ser dividido em sub-problemas, menores e mais simples, de modo a resolver cada sub-problema separadamente e unir a solução deles para resolver o problema maior, simplificando o processo.

Uma característica importante do uso de métodos é a possibilidade de isolar um trecho específico de código, que pode ser chamado de diferentes pontos do sistema, evitando a replicação de código. Isso garante flexibilidade e manutenibilidade ao software, uma vez que a mudança no processo implica na alteração do código em um ponto único.

A definição de um método consiste no modificador de acesso, tipo de retorno, identificador (nome), conjunto de argumentos e o conjunto de instruções do método: `<modificador> <tipo retorno> <identificador> ([<argumentos>]) {<instrucoes>}`. Por definição, um método deve resolver um problema específico, portanto possui um identificador intuitivo com relação à sua função.

### 1.9.1. Métodos sem retorno e sem parâmetros

A maneira mais simples em que um método se apresenta é quando não possui retorno e não recebe parâmetros. Neste caso, definimos seu retorno como `void` e não informamos qualquer informação na área de parâmetros (entre parêntesis). Dessa forma, a implementação do método ficaria como apresentado no código abaixo. A chamada do método é feita pelo seu identificador. No exemplo apresentado acima, não existe retorno nem passagem de parâmetros, portanto a chamada se limita à instrução com o identificador do método.

```
1 public void nomeMetodo() {  
2     System.out.println("Este código é executado quando o método é  
   ↳ chamado");  
3 }  
4  
5 //Chamada  
6 nomeMetodo();
```

### 1.9.2. Métodos com retorno e sem parâmetros

Frequentemente é desejável que o método retorne o resultado do seu processamento, de modo que possa ser utilizado pelo elemento que o invocou. Neste caso, deve-se definir o tipo de retorno e utilizar a cláusula `return` para retornar o valor desejado. O trecho de código abaixo ilustra a aplicação de um método com retorno.

```
1 public static void main(String[] args) {  
2     int resultado = nomeMetodo();  
3     System.out.println("O resultado retornado é: " + resultado);  
4 }  
5  
6 public static int nomeMetodo() {  
7     return 10;  
8 }
```

Neste exemplo, o método `nomeMetodo` retorna um valor constante (10). Este valor é recuperado no momento da chamada do método (linha 2) e posteriormente apresentado em tela (linha 3). Observe a necessidade da cláusula `static`, uma vez que o método é chamado pelo `main` (estático por definição).

### 1.9.3. Métodos com parâmetros

Finalmente, é possível enviar parâmetros ao método, os quais podem ser utilizados para o processamento desejado (estes métodos podem ser com ou sem retorno). Para isso, a definição do método contém a lista de parâmetros recebidos por ele. Esta lista é composta pelo tipo e identificador do parâmetro, separados por vírgula. O trecho de código abaixo exemplifica a definição e chamada de um método com passagem de parâmetros.

```
1 public static void main(String[] args) {
2     int resultado = nomeMetodo(50);
3     System.out.println("O resultado retornado é: " + resultado);
4 }
5
6 public static int nomeMetodo(int parametro) {
7     return parametro * 2;
8 }
```

Neste exemplo, o método `nomeMetodo` recebe um parâmetro do tipo inteiro (`parametro` – linha 4), retornando seu valor multiplicado por 2 (linha 7). A chamada do método (linha 2) passa como parâmetro o valor 50, recuperando seu resultado na variável `resultado`. Logo, o valor resultante, apresentado em tela na linha 3, possui o valor 100.

## 1.10. Vetores

Um vetor (também chamado de *array*) é um conjunto de variáveis do mesmo tipo, que possuem o mesmo identificador (nome) e se distinguem por um índice. Dessa forma, é possível armazenar um conjunto de valores através do mesmo identificador, isto é, uma lista de valores. Estas estruturas são armazenadas sequencialmente em memória.

### 1.10.1. Vetores unidimensionais

Um vetor pode ter uma ou muitas dimensões. No caso de uma dimensão, chamamos vetor unidimensional (ou simplesmente vetor) e o acessamos através de um índice simples. O trecho de código a seguir mostra a declaração e inicialização de valores de um vetor.

```
1 //Declaração de vetor de inteiros com 5 posições
2 int[] primeiroVetor = new int[5];
3
4 //Declaração e inicialização na mesma instrução
5 int[] segundoVetor = {1, 2, 3, 4, 5};
```

Para atribuir valores e recuperá-los da lista, utilizamos o índice para acessar cada posição. Um vetor de tamanho `n` é indexado de 0 até `n - 1`. O exemplo abaixo mostra a atribuição de um valor à posição 0 do vetor e a recuperação do valor armazenado na posição 2.

```
1 primeiroVetor[0] = 10; //Atribuição de valor
2 int valor = segundoVetor[2]; //Recuperação de valor
```

Para percorrer todas as posições de um vetor (para armazenar valores ou recuperá-los, por exemplo), deve-se utilizar um laço de repetição. O exemplo abaixo mostra a aplicação do laço `for` para realizar esta tarefa. Para isso, o laço inicia em 0 e vai até `n - 1`, sendo `n` o tamanho do vetor (a função `length` recupera o tamanho do vetor – linha 1). No

primeiro laço, são armazenados valores a cada posição do vetor. No segundo laço, estes valores são recuperados e apresentados em tela.

```
1  for(int i = 0; i < primeiroVetor.length; i++) {  
2      primeiroVetor[i] = i * 10;  
3  }  
4  
5  for(int i = 0; i < segundoVetor.length; i++) {  
6      System.out.println(segundoVetor[i]);  
7  }
```

## 1.10.2. Vetores multidimensionais

Ao utilizar mais de uma dimensão, chamamos vetor multidimensional. Um uso comum é utilizar duas dimensões, de modo a criar uma estrutura de tabela, chamada matriz. Neste caso, podemos armazenar valores às células da tabela e, portanto, acessamos cada célula através de dois índices (linha e coluna). O código abaixo mostra a criação e inicialização da matriz. Como a estrutura possui duas dimensões, devemos definir o tamanho de cada dimensão, isto é, o número de linhas e o número de colunas da matriz.

```
1  //Declaração de uma matriz de Strings 5x5  
2  String[] [] primeiraMatriz = new String[5][5];  
3  
4  //Declaração e inicialização  
5  String[] [] segundaMatriz = {{"a", "b"}, {"c", "d"}};
```

Para acessar os valores (na atribuição e na recuperação), devemos informar a linha e a coluna desejadas. O trecho de código abaixo ilustra estes casos.

```
1  primeiraMatriz[2][4] = "Texto";           //Atribuição de valor  
2  String letra = segundaMatriz[1][1];       //Recuperação de valor
```

Da mesma forma que fazemos com vetores unidimensionais, utilizamos laços de repetição para percorrer estas estruturas. No entanto, como temos duas dimensões, precisamos de dois laços de repetição aninhados, de modo a percorrer cada elemento da primeira e da segunda dimensões (respectivamente das linhas e colunas da matriz). O código abaixo mostra o comportamento supracitado, onde a `String ****` é atribuída a cada posição da matriz.

```
1  for(int i = 0; i < primeiraMatriz.length; i++) {  
2      for(int j = 0; j < primeiraMatriz.length; j++) {  
3          primeiraMatriz[i][j] = "****";  
4      }  
5  }
```

**Observação:** os mesmos conceitos se aplicam a matrizes com mais de duas dimensões.

## 1.11. Manipulação de String

Frequentemente valores textuais (**Strings**) precisam ser tratados (verificar se o usuário informou algum caracter não numérico, por exemplo). Neste caso, o Java fornece uma série de funções para manipulação de **Strings**.

Uma função muito útil é a `charAt(i)`, que retorna o caracter da posição `i` de uma **String**. O trecho de código abaixo mostra o funcionamento desta função, onde diferentes posições da variável `palavra` são acessadas por meio da função supracitada.

```
1 String palavra = "programação";
2 System.out.println(palavra.charAt(0)); // p
3 System.out.println(palavra.charAt(2)); // o
4 System.out.println(palavra.charAt(3)); // g
5 System.out.println(palavra.charAt(7)); // a
```

O código abaixo apresenta um exemplo, onde o caracter `'*'` é buscado dentro de uma **String**. Para isso, pode-se utilizar um laço de repetição, de modo a percorrer cada posição da **String**. Assim como com vetores, é possível recuperar o tamanho da **String** através da função `length`, para que se saiba o intervalo a ser percorrido pelo laço de repetição. O resultado da execução do código abaixo é: "Valor encontrado na posição 5".

```
1 String texto = "H8k#s*Opq.";
2 char busca = '*';
3 for(int i = 0; i < texto.length; i++) {
4     if(texto.charAt(i) == busca) {
5         System.out.println("Valor encontrado na posição " + i + "!");
6         break;
7     }
8 }
```

Existem várias outras funções para manipulação de **Strings** em Java. A Tabela 1.8 sumariza as principais delas e apresenta sua descrição.

Método	Descrição
<code>substring(int inicio, int fim)</code>	Retorna a substring a partir do caractere da posição <code>inicio</code> até a posição <code>fim</code> .
<code>equals(String s)</code>	Verifica se a <code>String</code> é igual a <code>s</code> .
<code>equalsIgnoreCase(String s)</code>	Idêntico ao anterior, mas sem diferenciar entre caracteres maiúsculos e minúsculos.
<code>contains(String s)</code>	Verifica se <code>s</code> é uma <code>substring</code> do objeto que chama a função.
<code>startsWith(String s)</code>	Verifica se a <code>String</code> inicia com a <code>substring</code> <code>s</code> .
<code>endsWith(String s)</code>	Verifica se a <code>String</code> termina com a <code>substring</code> <code>s</code> .
<code>toUpperCase()</code>	Transforma todos os caracteres em maiúsculos.
<code>toLowerCase()</code>	Transforma todos os caracteres em minúsculos.
<code>replace(String s1, String s2)</code>	Substitui os caracteres indicados em <code>s1</code> pelos indicados em <code>s2</code> .
<code>trim()</code>	Remove espaços no início e final da <code>String</code> .

Tabela 1.8.: Funções para manipulação de Strings em Java

## 2. Introdução à orientação a objetos

### 2.1. Visão geral

O paradigma orientado a objetos surgiu no final da década de 80. Alan Kay, um de seus idealizadores, formulou a chamada “analogia biológica”, onde propôs que um sistema de software deveria funcionar como um ser vivo. Neste sistema, cada célula interage com outras células através do envio de mensagens para realizar um objetivo comum. Adicionalmente, cada célula funciona como uma unidade autônoma. Neste sentido, o sistema de software é composto por um conjunto de unidades autônomas (os objetos), que interagem entre si para a troca de mensagens. A Figura 2.1 ilustra este conceito.

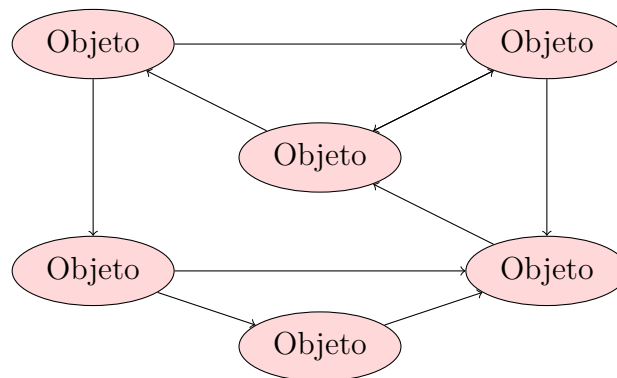


Figura 2.1.: Estrutura de um sistema orientado a objetos

A orientação a objetos possui alguns fundamentos básicos (apresentados abaixo), os quais serão utilizados e aplicados no restante deste material.

- Todo elemento do sistema é representado por um objeto.
- Objetos realizam tarefas através da requisição de serviços a outros objetos.
- Cada objeto pertence a uma determinada classe, que agrupa objetos similares.
- A classe define a estrutura e o comportamento dos seus objetos.
- Classes são organizadas em hierarquias.

### 2.2. Abstração

Abstrair significa focar nos elementos importantes de uma entidade ou processo, ignorando características e particularidades que não são de interesse a um determinado pro-

pósito. É importante destacar que a determinação do que é de interesse ou não depende do contexto.

Por exemplo, uma pessoa (entidade do mundo real) possui uma série de características, como o seu nome, idade, número de identidade, peso, altura, cor dos olhos, profissão e salário. Para o contexto de uma academia, são importantes as informações: nome, idade, peso e altura. Para o contexto de uma empresa de concessão de empréstimos, são importantes as informações: nome, idade, número de identidade, profissão e salário. Isto é, o processo de abstração é distinto para os dois contextos, pois para cada um, o conjunto de informações importantes é diferente.

## 2.3. Classes

Uma classe é uma representação conceitual e/ou computacional de uma entidade do mundo real. Ou seja, a classe é o resultado do processo de abstração de algo do mundo real. Portanto, uma classe é uma abstração das características dessa entidade. Neste processo de abstração, uma entidade é definida por sua estrutura (atributos e características) e pelo seu comportamento (funções que desempenha). A classe, portanto, define a estrutura e o comportamento de uma entidade. A estrutura é definida pelos atributos da classe, enquanto o comportamento é definido pelos métodos que a classe implementa.

Figura 2.2 mostra o processo de abstração para construção de uma classe. A entidade considerada é analisada, em busca da abstração da sua estrutura e comportamento. A estrutura é traduzida nos atributos da classe, enquanto o comportamento é traduzido nos seus métodos.

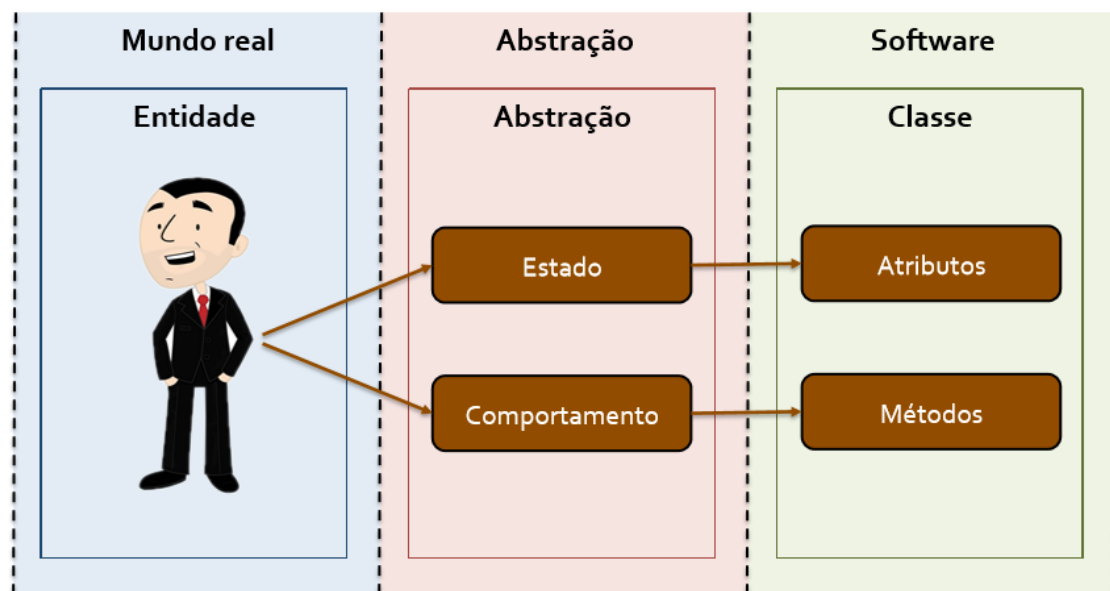


Figura 2.2.: Processo de abstração para construção de classes

A Figura 2.3 apresenta a representação de uma classe que modela uma pessoa. Esta representação segue os padrões definidos pela UML (Unified Modeling Language). Como pode ser observado, a classe divide-se em três partes. A primeira mostra o nome da



classe (Pessoa). Por padrão, o nome deve iniciar com letra maiúscula e seguir o formato camelCase. A segunda parte mostra os atributos da classe (sua estrutura), apresentando o nome de cada atributo, seu tipo e sua visibilidade (também chamado de modificador). Pode-se ainda apresentar um valor inicial para o atributo, como é feito com o atributo **sexo**. Finalmente, a terceira parte apresenta o conjunto de métodos da classe (seu comportamento), contendo o identificador do método, sua visibilidade, conjunto de parâmetros e tipo de retorno.

Pessoa
<ul style="list-style-type: none"><li>- id: int</li><li>- nome: String</li><li>- sexo: char = 'M'</li><li>- idade: int</li></ul>
<ul style="list-style-type: none"><li>+ fazerAniversario(): void</li></ul>

Figura 2.3.: Representação de uma classe usando UML

No exemplo apresentado pela Figura 2.3, o processo de abstração da entidade **Pessoa** resultou no conjunto de atributos composto pelo **id**, **nome**, **sexo** e **idade**, assim como o método de **fazerAniversario()**.

## 2.4. Atributos e Métodos

Um **atributo** consiste em uma propriedade nomeada de uma classe e descreve a faixa de valores (tipo de dado) que pode assumir. Conforme discutido na seção anterior, os atributos definem as características (estrutura) presentes nos objetos da classe e dependem do domínio em questão. Cada objeto pode armazenar valores nos seus atributos. O conjunto de valores armazenados em um determinado momento constituem o estado do objeto.

Os **métodos** determinam o comportamento do objeto, ou seja, como ele age e reage, suas modificações de estado e interações com outros objetos. Além disso, eles definem o conjunto de operações que o objeto pode realizar. Estas operações podem ser solicitadas por outros objetos, conforme seus modificadores de acesso.

## 2.5. Objetos

Objetos são instâncias de classes. Enquanto uma classe é uma abstração, um objeto é a manifestação concreta dessa abstração. Ou seja, é uma instância da entidade que possui um estado (valores dos seus atributos), comportamento (implementação dos seus métodos) e identidade. A Figura 2.4 mostra a classe **Pessoa** e um objeto desta classe (**maria**). Observe que o objeto pertence a uma classe (possui a estrutura e o comportamento definidos por ela) e apresenta valores nos seus atributos (estado).

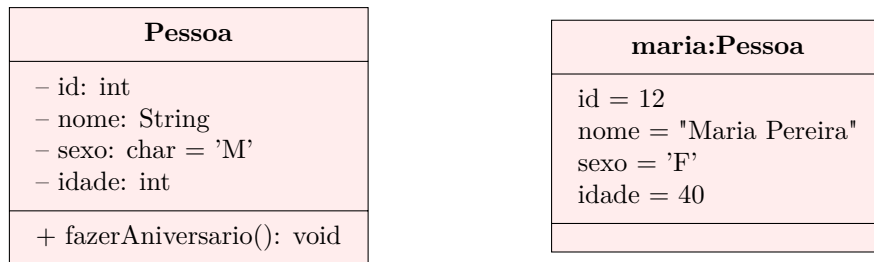


Figura 2.4.: Representação de uma classe e o respectivo objeto

## 2.6. Implementação de classes

Considerando o contexto de uma concessionária (sistema para gestão dos veículos), a classe **Veiculo** deve possuir: modelo, marca, ano, cor, potência e ar condicionado. Além disso, ela deve implementar um método para cálculo do imposto em função do ano do veículo. A Figura 2.5 apresenta a classe **Veiculo** com os atributos e método supracitados.

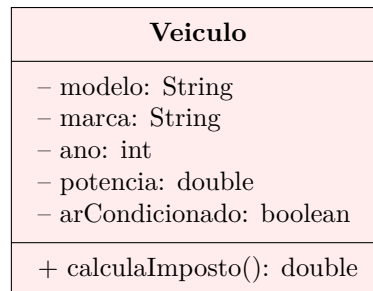


Figura 2.5.: Representação UML da classe “Veiculo”

O trecho de código abaixo mostra a implementação da classe apresentada pela Figura 2.5. A definição é feita pelas instruções `public class <nome-da-classe>`. Os atributos estão definidos nas linhas 3 a 7. Para cada atributo é definida sua visibilidade, seu tipo e o identificador (ex: `private String modelo`). O método da classe é definido nas linhas 9 a 13.

```

1  public class Veiculo {
2
3      private String modelo;
4      private String marca;
5      private int ano;
6      private double potencia;
7      private boolean arCondicionado;
8
9      public double calculaImposto() {
10         if(ano < 2010)
11             return 500d;
12         return 700d;
13     }

```

14 }

## 2.7. Encapsulamento

Encapsular significa agrupar e empacotar os detalhes internos da abstração, tornando-os inacessíveis para entidades externas. Cada classe define a sua interface, composta pelos atributos e métodos que podem ser acessados/chamados a partir de objetos externos à classe. Neste sentido, encapsular consiste em definir quais elementos devem estar disponíveis a entidades externas, empacotando os detalhes daquilo que não deve estar disponível. A Figura 2.6 apresenta uma representação gráfica do conceito de encapsulamento.

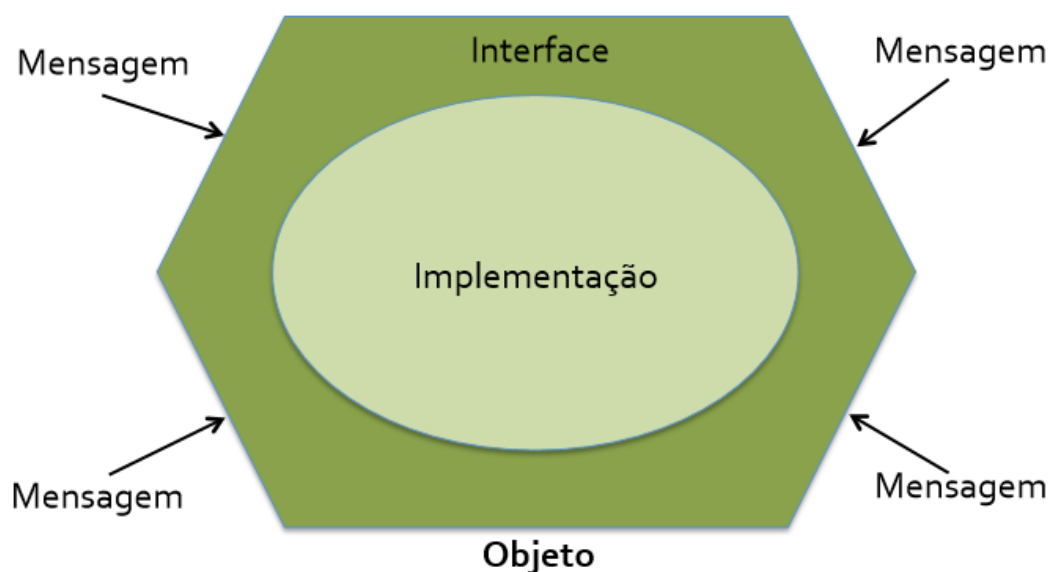


Figura 2.6.: Visão geral do conceito de encapsulamento

Existem três principais modificadores de acesso:

- **Privado (-):** O elemento é acessado apenas pela classe que o define.
- **Protegido (#):** O elemento é acessado pela classe que o define e também por suas sub-classes.
- **Público (+):** Qualquer objeto pode acessar o elemento.

Os trechos de código abaixo exemplificam o uso do encapsulamento no atributo `saldo` da classe `ContaCorrente`. Na classe sem encapsulamento, o saldo pode ser acessado diretamente. Ou seja, o valor de saldo pode ser modificado por qualquer entidade externa (para a posterior realização de um saque, por exemplo). Este problema é corrigido na segunda classe, encapsulando o atributo `saldo`.

```
1 //Classe sem encapsulamento
2 public class ContaBancaria {
3     public double saldo;
```

```
4
5     public boolean saque(double valor) {
6         if(saldo >= valor) {
7             saldo -= valor;
8             return true;
9         }
10        return false;
11    }
12 }
```

```
1 //Classe com encapsulamento
2 public class ContaBancaria {
3     private double saldo;
4
5     public boolean saque(double valor) {
6         if(saldo >= valor) {
7             saldo -= valor;
8             return true;
9         }
10        return false;
11    }
12 }
```

## 2.8. Métodos acessores

Por conta do encapsulamento, os atributos das classes devem ser privados. Os métodos assessores (também conhecidos por métodos `get` e `set`) são responsáveis por devolver os valores dos atributos ou alterar seus valores, quando solicitado.

Os métodos `get` (ex: `getNome()`) são responsáveis por devolver o valor de um atributo do objeto. Geralmente não possuem argumentos e possuem um tipo de retorno (o tipo do atributo). São públicos por padrão.

Os métodos `set` (ex: `setNome(String nome)`) são responsáveis por realizar a modificação dos valores dos atributos. Geralmente possuem como argumento o valor a ser atribuído e não possuem retorno. São públicos por padrão.

O trecho de código abaixo mostra o uso dos métodos acessores para acesso (recuperação e atribuição de valores) ao atributo `modelo` da classe `Veiculo`. Na linha 5, o argumento recebido pelo método `set` é armazenado no atributo `modelo`. Já na linha 9, o valor armazenado no atributo `modelo` é retornado pelo método `get`.

```
1 public class Veiculo {
2     private String modelo;
3
4     public void setModelo(String modelo) {
```

```
5     this.modelo = modelo;
6 }
7
8 public String getModelo() {
9     return this.modelo;
10 }
11 }
```

## 2.9. Métodos construtores

Quando um objeto é criado (ex: `pessoa = new Pessoa();`), um conjunto de passos é executado:

1. Inicialização default dos campos (`null`, `false`, `0`).
2. Chamada recursiva aos construtores de cada superclasse (até `Object`).
  - a) Inicialização default dos campos das superclasses.
  - b) Execução do conteúdo dos métodos construtores de cada superclasse (desde `Object`).
3. Execução do conteúdo do método construtor da classe.

Logo, o construtor é um método onde podemos definir o que será realizado quando um objeto da classe é criado. O método construtor não possui tipo de retorno e seu identificador é idêntico ao nome da classe. O trecho de código abaixo mostra a definição do método construtor da classe `Pessoa` (método `Pessoa()`). Com isso, cada objeto de `Pessoa` criado possui o valor `sem nome` no atributo `nome` e `1` no atributo `idade`.

```
1 public class Pessoa {
2
3     private String nome;
4     private int idade;
5
6     public Pessoa() {
7         //Neste exemplo, são definidos valores
8         //iniciais para cada objeto criado
9         this.nome = "sem nome";
10        this.idade = 1;
11    }
12 }
```

É possível (e comum) definir argumentos no método construtor, que são passados no momento em que o objeto é criado. Com isso, valores já podem ser armazenados nos atributos da instância. O trecho de código abaixo exemplifica a definição de um método construtor parametrizado para a classe `Pessoa`. Com isso, os valores recebidos como parâmetros são armazenados nos atributos da classe.

```
1 public class Pessoa {
2
3     private String nome;
4     private int idade;
5
6     public Pessoa() {
7         this.nome = "Não especificado";
8         this.idade = 1;
9     }
10
11     public Pessoa(String nome, int idade) {
12         this.nome = nome;
13         this.idade = idade;
14     }
15 }
```

Com isso, é possível definir dois ou mais métodos construtores, sem parâmetros ou com parâmetros. Esta técnica é chamada **sobrecarga de operações** e é discutida na Seção 2.11. Quando não especificado um método construtor, o Java fornece um construtor padrão (sem argumentos e sem implementação). A partir do momento em que o programador define um método construtor, o construtor padrão deixa de existir.

**OBS:** o uso da instrução `this` serve para acessar a instância do objeto atual. É usado para diferenciar os atributos de variáveis locais.

## 2.10. Métodos destrutores

No Java, os objetos são destruídos (eliminados da memória) automaticamente, quando não existe mais nenhum ponteiro para ele. É possível definir um método a ser executado na destruição do objeto. Este método é o `finalize()` e suas instruções são executadas imediatamente antes da destruição do objeto. O trecho de código abaixo exemplifica seu uso.

```
1 public class Pessoa {
2     private String nome;
3     private int idade;
4
5     public void finalize(){
6         System.out.println("Objeto destruído: " + this.getNome());
7     }
8 }
```

## 2.11. Sobrecarga de operações

Assim como os métodos construtores, métodos comuns também podem ser sobrecarregados. Ou seja, pode-se criar dois ou mais métodos com o mesmo identificador, desde que seus parâmetros sejam diferentes. Em outras palavras, a assinatura do método deve ser única. A assinatura é definida pelo identificador do método e os tipos dos seus parâmetros. O trecho de código a seguir mostra a sobrecarga do construtor da classe e também do método `soma`.

```
1 public class Acumulador{
2     private long somaInteiros;
3
4     public Acumulador() {
5         this.somaInteiros = 0;
6     }
7
8     public Acumulador(long valorInicial){
9         this.somaInteiros = valorInicial;
10    }
11
12    public void soma(int valor){
13        this.somaInteiros += valor;
14    }
15
16    public void soma(String valor) {
17        this.somaInteiros += Integer.parseInt(valor);
18    }
19 }
```

No código anterior, os métodos `soma` possuem as assinaturas `void soma(int)` e `void soma(String)`, respectivamente. Dessa forma, o Java sabe qual dos dois métodos chamar, de acordo com o argumento passado a ele. Um terceiro método com um argumento do tipo `String` não seria permitido, pois sua assinatura seria idêntica a um dos métodos sobrecarregados. Ou seja, o Java não saberia qual dos dois chamar.

## 2.12. Exemplo – Compra

Considere um sistema para gerenciar compras feitas pela Internet. O sistema deve armazenar os dados dos produtos comprados. A entidade envolvida neste sistema é apresentada pela Figura 2.7.

O trecho de código abaixo mostra a implementação da classe `Compra`. As linhas 3 a 7 implementam os atributos da classe, com modificador `private`. Ou seja, os atributos estão encapsulados e seu acesso é permitido através dos métodos acessores, também apresentados nos códigos abaixo. A classe ainda define dois métodos construtores: um vazio, e também sem implementação, e um parametrizado (linhas 9 a 18). Finalmente, a classe

Compra
<ul style="list-style-type: none"><li>– produto: String</li><li>– precoUnitario: double</li><li>– quantidade: int</li><li>– enderecoEntrega: String</li><li>– status: String</li></ul>
<ul style="list-style-type: none"><li>+ métodos construtores</li><li>+ métodos set() e get()</li><li>+ valorTotal(): double</li></ul>

Figura 2.7.: Representação UML da classe “Compra”

implementa um método `valorTotal`, que multiplica o preço unitário pela quantidade (linhas 20 a 22).

```
1  public class Compra {
2
3      private String produto;
4      private double precoUnitario;
5      private int quantidade;
6      private String enderecoEntrega;
7      private String status;
8
9      public Compra() {}
10
11     public Compra(String produto, double precoUnitario, int
12 ↪ quantidade, String enderecoEntrega, String status) {
13
14         this.produto = produto;
15         this.precoUnitario = precoUnitario;
16         this.quantidade = quantidade;
17         this.enderecoEntrega = enderecoEntrega;
18         this.status = status;
19     }
20
21     public double valorTotal() {
22         return this.precoUnitario * this.quantidade;
23     }
24
25     public String getProduto() {
26         return produto;
27     }
28
29     public void setProduto(String produto) {
30         this.produto = produto;
31     }
32 }
```



```
32     public double getPrecoUnitario() {
33         return precoUnitario;
34     }
35
36     public void setPrecoUnitario(double precoUnitario) {
37         this.precoUnitario = precoUnitario;
38     }
39
40     public int getQuantidade() {
41         return quantidade;
42     }
43
44     public void setQuantidade(int quantidade) {
45         this.quantidade = quantidade;
46     }
47
48     public String getEnderecoEntrega() {
49         return enderecoEntrega;
50     }
51
52     public void setEnderecoEntrega(String enderecoEntrega) {
53         this.enderecoEntrega = enderecoEntrega;
54     }
55
56     public String getStatus() {
57         return status;
58     }
59
60     public void setStatus(String status) {
61         this.status = status;
62     }
63 }
```

O trecho de código abaixo mostra a implementação da classe `ExemploCompra`, que utiliza a classe `Compra` para suas operações. Nesta classe está a lista que armazena as compras efetuadas (`compras` – linha 3). No método `main` são criados dois objetos da classe `Compra`. Na primeira vez, o objeto é instanciado e seus valores são atribuídos através dos métodos acessores (linhas 6 a 11). Na segunda vez, o objeto é criado e seus valores são passados no momento da criação, utilizando o construtor parametrizado (linha 15). Ambos os objetos, após criados, são inseridos na lista de compras (linhas 13 e 17). Finalmente, a lista é percorrida e os objetos são exibidos em tela (linhas 19 a 22).

```
1     public class ExemploCompra {
2
3         private static List<Compra> compras = new ArrayList<Compra>();
4
5         public static void main(String[] args) {
```

```
6      Compra compra = new Compra();
7      compra.setProduto("Camiseta XPTO - TAM M");
8      compra.setPrecoUnitario(140.0);
9      compra.setQuantidade(2);
10     compra.setEnderecoEntrega("Rua Getúlio Vargas, 200 - Ibirama,
↪ SC - 89140-000");
11     compra.setStatus("Pedido realizado");
12
13     compras.add(compra);
14
15     Compra compra2 = new Compra("Tênis ABC 42", 300.0, 1, "Caixa
↪ Postal 27 - 89140-970", "Pagamento confirmado");
16
17     compras.add(compra2);
18
19     for(int i = 0; i < compras.size(); i++) {
20         Compra c = compras.get(i);
21         System.out.println("Compra: " + c.getProduto() + " a " +
↪ c.getPrecoUnitario() + "/unidade, totalizando " + c.valorTotal()
↪ + ".");
22     }
23 }
24 }
```

**SAÍDA:**

Compra: Camiseta XPTO - TAM M a 140.0/unidade, totalizando 280.0.

Compra: Tênis ABC 42 a 300.0/unidade, totalizando 300.0.

## **Parte II.**

### **Relacionamentos entre classes**



## 3. Visão geral

### 3.1. Tipos de relacionamentos

As classes podem se conter relacionamentos entre si, definindo um vínculo entre seus objetos. Considerando as entidades em negrito, alguns exemplos de relacionamentos entre classes incluem:

- Um **cliente** possui um **endereço**.
- Uma **empresa** é composta por **funcionários**.
- Uma **moto** é um tipo de **veículo**.
- Um **restaurante** possui **pratos**.
- Uma **correspondência** possui um **remetente** e um **destinatário**.

Existem diferentes tipos de relacionamentos. A lista abaixo resume os principais tipos de relacionamentos, os quais serão detalhados neste capítulo.

- **Associação:** conexão entre classes.
- **Agregação e composição:** especialização de uma associação onde um todo é relacionado com suas partes (relacionamento “todo-parte”).
- **Dependência:** um objeto depende de alguma forma de outro (relacionamento de utilização).
- **Herança (generalização):** um dos princípios da orientação a objetos, onde uma nova classe pode ser definida a partir de outra já existente (reutilização).
- **Realização:** um contrato que classe segue (obrigação).

### 3.2. Representação UML

A Figura 3.1 mostra a representação UML dos relacionamentos supracitados. Uma associação é representada por um segmento de reta que une as duas classes. Os relacionamentos de agregação e composição são representados por uma reta com um losango no lado da entidade que representa o todo. A composição se difere da primeira por apresentar o losango preenchido. A dependência é representada por uma linha tracejada com uma ponta de seta aberta no lado da classe independente. A herança é representada por uma linha com uma ponta de seta fechada no lado da superclasse. Finalmente, a realização

é representada da mesma forma que uma herança, com a diferença da reta ser tracejada. O restante deste capítulo discute cada relacionamento entre classes e apresenta sua implementação utilizando a linguagem Java.

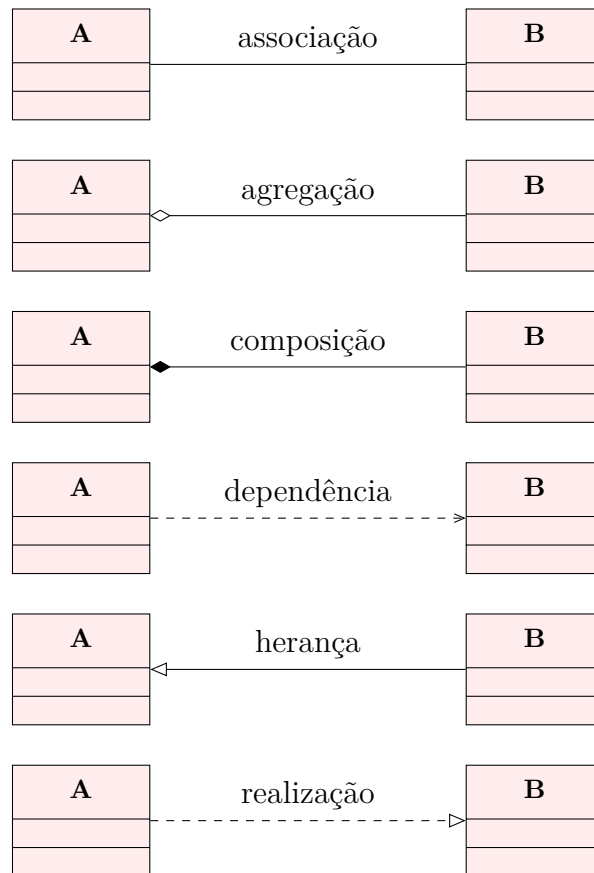


Figura 3.1.: Representação UML dos relacionamentos entre classes

## 4. Associações simples

Uma associação é uma conexão entre classes e representa uma relação entre os objetos envolvidos. Elas são representadas em um diagrama de classes através de uma linha, que conecta as classes associadas. Os dados podem fluir em uma ou em ambas as direções através do link. Existe duas formas principais de associação: com multiplicidade um (1) e com multiplicidade muitos (\*). Cada uma das formas de associação pode ser implementada de forma unidirecional ou bidirecional.

### 4.1. Associação com multiplicidade um (1)

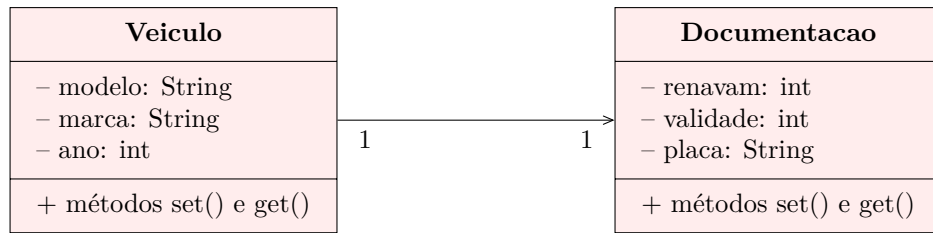
Uma associação de multiplicidade 1 ocorre quando um objeto está vinculado a apenas um objeto da outra classe. Um exemplo deste tipo de associação é o vínculo existente entre um veículo (primeira entidade – classe) e sua documentação (segunda entidade – classe). Um veículo possui exatamente uma documentação, enquanto uma documentação pertence a exatamente um veículo. Logo, dizemos que este relacionamento é uma **associação de um para um**.

Na associação com multiplicidade 1, ao menos um dos atributos de uma das classes é do tipo da outra classe. Consideremos um exemplo envolvendo as classes **Veiculo** e **Documentacao**. Neste caso, o **Veiculo** possui um atributo do tipo **Documentacao** (**associação unidirecional de Veiculo para Documentacao**), ou a **Documentacao** possui um atributo do tipo **Veiculo** (**associação unidirecional de Documentacao para Veiculo**), ou ambos os casos (**associação bidirecional**).

#### 4.1.1. Caso unidirecional

Consideremos o caso unidirecional. Se queremos que o atributo que implementa a associação fique na classe **Veiculo**, então temos uma associação unidirecional de **Veiculo** para **Documentacao**. Sua representação UML é apresentada na Figura 4.1. A direção da associação (chamada de navegabilidade) é informada através de uma ponta de seta. A navegabilidade representada na Figura 4.1 mostra que a partir da classe **Veiculo** podemos chegar na classe **Documentacao**, pois existe um atributo na classe **Veiculo** do tipo **Documentacao**. Percebam que o atributo que implementa a associação não é representado no diagrama de classes, pois é implícito na representação da associação.

**OBS:** ao dizer que o veículo possui exatamente uma documentação, o vínculo é obrigatório. Ou seja, o atributo **documentacao** da classe **Veiculo** deve, obrigatoriamente, receber uma referência a um objeto da classe **Documentacao**. Caso quiséssemos modelar o relacionamento de tal forma que o veículo pudesse existir mesmo sem o vínculo com a documentação, então a multiplicidade do lado **Documentacao** deveria ser **0..1**.

Figura 4.1.: Associação unidirecional de **Veiculo** para **Documentacao**

Os trechos de código abaixo mostram a implementação das classes apresentadas no diagrama da Figura 4.1 (os métodos construtores e demais métodos acessores foram omitidos). Reparem o atributo `doc` na classe **Veiculo**. Assim como os demais atributos, métodos acessores são definidos para acesso à documentação do veículo (ao atributo `doc`).

```
1 public class Veiculo {
2     private String modelo;
3     private String marca;
4     private int ano;
5     private Documentacao doc;
6
7     public Documentacao getDoc() {
8         return doc;
9     }
10
11    public void setDoc(Documentacao doc) {
12        this.doc = doc;
13    }
14
15    //demais métodos
16 }
```

```
1 public class Documentacao {
2     private int renavam;
3     private int validade;
4     private String placa;
5
6     public int getRenavam() {
7         return renavam;
8     }
9
10    public void setRenavam(int renavam) {
11        this.renavam = renavam;
12    }
13
14    //demais métodos
```



15 }

O trecho de código a seguir mostra o uso da associação. Uma vez que as classes estejam associadas, devemos vincular os objetos no momento da sua criação (linha 4). Além disso, podemos acessar o vínculo para recuperar valores da documentação, a partir de um objeto da classe `Veiculo` (linha 5). Repare que não é possível acessar os dados do veículo a partir de um objeto da classe `Documentacao`, uma vez que o atributo que realiza a associação encontra-se na classe `Veiculo`.

```

1 public static void main(String[] args) {
2     Veiculo carro = new Veiculo("Focus", "Ford", 2017);
3     Documentacao doc = new Documentacao(512647522, 2018, "QWD-2573");
4     carro.setDoc(doc);
5     System.out.println("O veículo " + carro.getModelo() + " possui
   ↳ placa " + carro.getDoc().getPlaca());
6 }

```

Ainda no caso unidirecional, se quisermos que o atributo que implementa a associação fique na classe `Documentacao`, então temos uma associação unidirecional de `Documentacao` para `Veiculo`. Sua representação UML é apresentada na Figura 4.2. A direção da associação (navegabilidade) é apresentada pela ponta de seta, que aponta para a classe `Veiculo`. Neste caso, a partir da classe `Documentacao` podemos chegar na classe `Veiculo`, pois existe um atributo na classe `Documentacao` do tipo `Veiculo`.

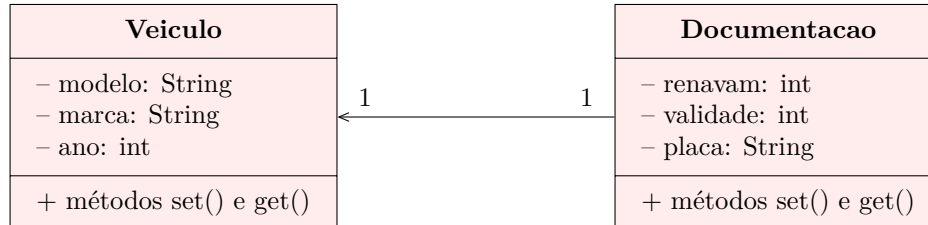


Figura 4.2.: Associação unidirecional de `Documentacao` para `Veiculo`

**OBS:** novamente, ao dizer que a documentação é de exatamente um veículo, o vínculo é obrigatório. Ou seja, o atributo `veiculo` da classe `Documentacao` deve, obrigatoriamente, receber uma referência a um objeto da classe `Veiculo`. Caso quiséssemos modelar o relacionamento de tal forma que a documentação pudesse existir mesmo sem o vínculo com o veículo, então a multiplicidade do lado `Veiculo` deveria ser `0..1`.

Os trechos de código abaixo mostram a implementação das classes apresentadas no diagrama da Figura 4.2 (os métodos construtores e demais métodos acessores foram omitidos). Reparem o atributo `veiculo` na classe `Documentacao`. Assim como os demais atributos, métodos acessores são definidos para acesso ao veículo da documentação (ao atributo `veiculo`).

```

1 public class Veiculo {
2     private String modelo;
3     private String marca;

```

```
4     private int ano;
5
6     public String getModelo() {
7         return modelo;
8     }
9
10    public void setModelo(String modelo) {
11        this.modelo = modelo;
12    }
13
14    //demais métodos
15 }
```

```
1 public class Documentacao {
2     private int renavam;
3     private int validade;
4     private String placa;
5     private Veiculo veiculo;
6
7     public Veiculo getVeiculo() {
8         return veiculo;
9     }
10
11    public void setVeiculo(Veiculo veiculo){
12        this.veiculo = veiculo;
13    }
14
15    //demais métodos
16 }
```

O trecho de código a seguir mostra o uso da associação. Analogamente ao exemplo anterior, podemos acessar o vínculo para recuperar valores do veículo a partir de um objeto da classe `Documentacao` (linha 5). Repare que não é possível acessar os dados da documentação a partir de um objeto da classe `Veiculo`, uma vez que o atributo que realiza a associação encontra-se na classe `Documentacao`.

```
1 public static void main(String[] args) {
2     Veiculo carro = new Veiculo("Focus", "Ford", 2017);
3     Documentacao doc = new Documentacao(512647522, 2018, "QWD-2573");
4     doc.setVeiculo(carro);
5     System.out.println("O documento " + doc.getRenavam() + " pertence
↪ ao veículo " + doc.getVeiculo().getModelo());
6 }
```

### 4.1.2. Caso bidirecional

Se quisermos que ambas as classes possuam atributos que implementem o vínculo, podemos adotar uma associação bidirecional. Neste caso, existe um atributo da classe **Veiculo** na classe **Documentacao**, bem como um atributo da classe **Documentacao** na classe **Veiculo**. Com isso, é possível acessar a documentação a partir da classe **Veiculo**, bem como acessar o veículo a partir da classe **Documentacao**. Na UML, representamos uma associação bidirecional aplicando a ponta de seta nos dois lados da linha, ou omitindo a ponta de seta e mantendo uma linha simples (neste caso, entende-se que a associação é bidirecional). A Figura 4.3 mostra as duas opções de representação UML de um relacionamento de associação bidirecional.

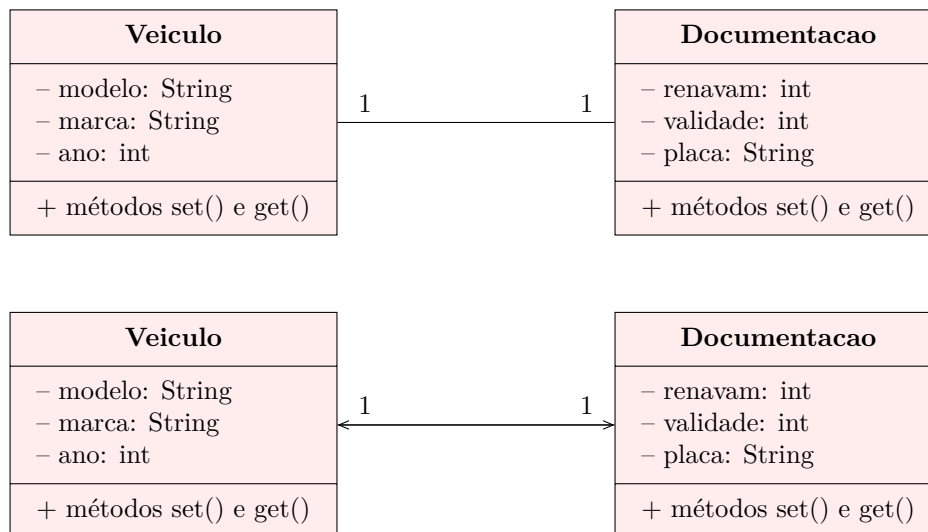


Figura 4.3.: Associação unidirecional de Documentacao para Veiculo

Os trechos de código abaixo mostram a implementação da associação bidirecional entre **Veiculo** e **Documentacao**. Repare nos atributos **doc** (classe **Veiculo**) e **veiculo** (classe **Documentacao**), os quais implementam o vínculo bidirecional e permite a navegação em ambos os sentidos.

```

1  public class Veiculo {
2      private String modelo;
3      private String marca;
4      private int ano;
5      private Documentacao doc;
6
7      public Documentacao getDoc() {
8          return doc;
9      }
10
11     public void setDoc(Documentacao doc) {
12         this.doc = doc;
13     }
14

```

```
15    //demais métodos
16 }

1 public class Documentacao {
2     private int renavam;
3     private int validade;
4     private String placa;
5     private Veiculo veiculo;
6
7     public Veiculo getVeiculo() {
8         return veiculo;
9     }
10
11    public void setVeiculo(Veiculo veiculo){
12        this.veiculo = veiculo;
13    }
14
15    //demais métodos
16 }
```

O código a seguir mostra o uso das entidades com a associação bidirecional. Após a criação dos objetos, eles são vinculados nas linhas 4 e 5. Perceba que é preciso atribuir o veículo à sua documentação, e também a documentação ao seu veículo, uma vez que a associação é bidirecional. Após isso, é possível acessar os dados do veículo a partir da documentação (linha 6), bem como acessar os dados da documentação a partir do veículo (linha 7).

```
1 public static void main(String[] args) {
2     Veiculo carro = new Veiculo("Focus", "Ford", 2017);
3     Documentacao doc = new Documentacao(512647522, 2018, "QWD-2573");
4     doc.setVeiculo(carro);
5     carro.setDoc(doc);
6     System.out.println(doc.getRenavam() + " pertence ao " +
↪ doc.getVeiculo().getModelo());
7     System.out.println(carro.getModelo() + " possui placa " +
↪ carro.getDoc().getPlaca());
8 }
```

## 4.2. Exemplo – Gerente e Sala

Consideremos as entidades **Gerente** e **Sala**. Cada gerente possui uma sala de trabalho. Cada sala é ocupada por, no máximo, um gerente. Logo, as entidades possuem uma associação, cuja multiplicidade máxima é 1. No entanto, neste caso um gerente pode não trabalhar em uma sala (caso seja um gerente externo, por exemplo). Além disso, uma sala pode não possuir nenhum gerente que trabalhe nela (uma sala vazia, por exemplo).

Podemos informar a possibilidade de haver estes casos no próprio diagrama de classes, atribuindo multiplicidade `0..1`, ou seja, um gerente possui zero ou uma sala. Com isso, a chamada para vínculo dos objetos não é obrigatória e o atributo que implementa a associação pode não receber um vínculo (armazenar valor `null`).

Se desejarmos que a entidade **Gerente** tenha acesso aos dados da sua sala e, ao mesmo tempo, a entidade **Sala** tenha acesso aos dados do gerente que trabalha nela, podemos optar por uma associação bidirecional. O diagrama apresentado pela Figura 4.4 modela esta situação. As classes possuem uma associação bidirecional com multiplicidade `0..1` nos dois lados. Com isso, um gerente trabalha em uma sala ou em nenhuma sala, enquanto uma sala é habitada por um gerente, ou por nenhum gerente.

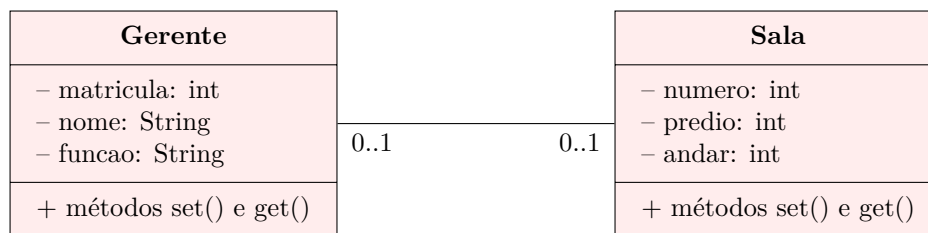


Figura 4.4.: Diagrama de classes para o exemplo Sala e Gerente

Os techos de código abaixo mostram a implementação das classes **Gerente** e **Sala**. Os métodos `setSala` e `setGerente` foram implementados de tal forma que permita a chamada de uma única instrução, sem exigir que o programador faça o vínculo dos dois lados. Com isso, o programador se preocupa apenas em chamar o método `setSala` (da classe **Gerente**), que armazena o objeto recebido como parâmetro no atributo `sala` (linha 27) e, caso o parâmetro não seja `null`, atribui a si mesmo como gerente da sala (linhas 28 a 30), fazendo o vínculo nos dois lados. Já o método `setGerente` verifica se o parâmetro recebido não é `null`, neste caso o gerente antes vinculado a esta sala recebe `null` como sala, pois um novo gerente trabalhará nesta sala (linha 21). Finalmente, o novo gerente é atribuído (linha 22).

```

1  public class Gerente {
2      private int matricula;
3      private String nome;
4      private String funcao;
5      private Sala sala;
6
7      public Gerente() {}
8
9      public Gerente(int matricula, String nome, String funcao) {
10         this.matricula = matricula;
11         this.nome = nome;
12         this.funcao = funcao;
13     }
14
15     public Gerente(int matricula, String nome, String funcao, Sala
        ↪ sala) {
  
```

```
16     this.matricula = matricula;
17     this.nome = nome;
18     this.funcao = funcao;
19     setSala(sala);
20 }
21
22 public Sala getSala() {
23     return sala;
24 }
25
26 public void setSala(Sala sala) {
27     this.sala = sala;
28     if(sala != null)
29         sala.setGerente(this);
30 }
31 }
32
33 //demais métodos acessores
34 }
```

```
1 public class Sala {
2     private int numero;
3     private String predio;
4     private int andar;
5     private Gerente gerente;
6
7     public Sala() {}
8
9     public Sala(int numero, String predio, int andar) {
10         this.numero = numero;
11         this.predio = predio;
12         this.andar = andar;
13     }
14
15     public Gerente getGerente() {
16         return gerente;
17     }
18
19     public void setGerente(Gerente gerente) {
20         if(this.gerente != null)
21             this.gerente.setSala(null);
22         this.gerente = gerente;
23     }
24 }
25
26 //demais métodos acessores
```

27 }  
}

Uma boa prática consiste em separar as funcionalidades do programa da classe que implementa o método `main`. Nos exemplos abaixo, o método `main` é implementado pela classe `Principal`, enquanto as funcionalidades do sistema são implementadas pela classe `Exemplo`. Esta classe ainda implementa um método `run`, que chama os métodos e as instruções desejadas (poderia apresentar um menu ao usuário, por exemplo). O método `main` apenas cria um objeto da classe `Exemplo` e chama o seu método `run`. Ou seja, a função do método `main` se limita a iniciar a aplicação.

```
1 public class Principal {
2     public static void main(String[] args) {
3         Exemplo exemplo = new Exemplo();
4         exemplo.run();
5     }
6 }

1 public class Exemplo {
2     private List<Sala> salas = new ArrayList<Sala>();
3     private List<Gerente> gerentes = new ArrayList<Gerente>();
4
5     public void run() {
6         insereSalas();
7         insereGerentes();
8         mostraRegistros();
9     }
10
11    public void insereSalas() {
12        Sala sala1 = new Sala(101, "Alpha", 1);
13        Sala sala2 = new Sala(102, "Alpha", 1);
14        Sala sala3 = new Sala(205, "Alpha", 2);
15        Sala sala4 = new Sala(346, "Beta", 5);
16        Sala sala5 = new Sala(12, "Gamma", 3);
17
18        salas.add(sala1);
19        salas.add(sala2);
20        salas.add(sala3);
21        salas.add(sala4);
22        salas.add(sala5);
23    }
24
25    public void insereGerentes() {
26        Gerente gerente1 = new Gerente(123456, "José da Silva",
↪ "Compras");
27        Gerente gerente2 = new Gerente(654321, "Maria Pereira",
↪ "Vendas");
```

```

28     Gerente gerente3 = new Gerente(123789, "João Assunção",
↪     "Marketing");
29     Gerente gerente4 = new Gerente(987321, "Ana Maria Rodrigues",
↪     "Produção");
30
31     gerente1.setSala(salas.get(0));
32     gerente2.setSala(salas.get(1));
33     gerente3.setSala(salas.get(2));
34     gerente4.setSala(salas.get(0));
35
36     gerentes.add(gerente1);
37     gerentes.add(gerente2);
38     gerentes.add(gerente3);
39     gerentes.add(gerente4);
40 }
41
42 public void mostraRegistros() {
43     for(Gerente g : gerentes) {
44         if(g.getSala() != null)
45             System.out.println(g.getNome() + " trabalha na sala " +
↪ g.getSala().getNumero() + " do prédio " + g.getSala().getPredio()
↪ + ".");
46         else
47             System.out.println(g.getNome() + " não possui uma sala de
↪ trabalho.");
48     }
49
50     System.out.println("");
51
52     for(Sala s : salas) {
53         if(s.getGerente() != null)
54             System.out.println("Na sala " + s.getNumero() + " do
↪ prédio " + s.getPredio() + " trabalha " +
↪ s.getGerente().getNome() + ".");
55         else
56             System.out.println("Na sala " + s.getNumero() + " do
↪ prédio " + s.getPredio() + " não trabalha nenhum gerente.");
57     }
58 }
59 }

```

O método `insereSalas` cria um conjunto de salas e insere na lista de salas. O método `insereGerentes` cria um conjunto de gerentes, vincula a algumas das salas criadas pelo método anterior e os insere na lista de gerentes. Repare que o vínculo é feito apenas pelo método `setSala`, dada a implementação apresentada anteriormente. O método `mostraRegistros` percorre a lista de gerentes e apresenta seu nome, a sala e o prédio onde trabalha. Caso ele não esteja vinculado a nenhuma sala (quando o atributo `sala` é `null`), é apresentada a mensagem de que o mesmo não possui sala de trabalho. Depois disso,



a lista de salas é percorrida, apresentando o número da sala e seu prédio, juntamente com o nome do gerente que ocupa esta sala. Caso a sala esteja vazia (o atributo `gerente` é `null`), é apresentada a mensagem de que na referida sala não trabalha nenhum gerente.

#### SAÍDA:

```
José da Silva não possui uma sala de trabalho.  
Maria Pereira trabalha na sala 102 do prédio Alpha.  
João Assunção trabalha na sala 205 do prédio Alpha.  
Ana Maria Rodrigues trabalha na sala 101 do prédio Alpha.
```

```
Na sala 101 do prédio Alpha trabalha Ana Maria Rodrigues.  
Na sala 102 do prédio Alpha trabalha Maria Pereira.  
Na sala 205 do prédio Alpha trabalha João Assunção.  
Na sala 346 do prédio Beta não trabalha nenhum gerente.  
Na sala 12 do prédio Gamma não trabalha nenhum gerente.
```

### 4.3. Associação com multiplicidade muitos (\*)

A associação com multiplicidade muitos (\*) ocorre quando um objeto está vinculado a vários objetos da outra classe. Um exemplo deste tipo de associação é o vínculo existente entre uma empresa (primeira entidade – classe) e seus funcionários (segunda entidade – classe). Um funcionário trabalha em uma empresa, e uma empresa possui vários (ou muitos) funcionários. Perceba que, agora, um objeto da classe `Empresa` está associado a vários objetos da classe `Funcionario`. Logo, dizemos que este relacionamento é uma **associação de um para muitos**.

Como em qualquer associação, os dados podem fluir em uma ou em ambas as direções através do link. Isto é, podemos implementar uma associação com multiplicidade muitos tanto na forma unidirecional, quanto bidirecional. Podemos utilizar um atributo da classe `Empresa` na classe `Funcionario` (**associação unidirecional de Funcionario para Empresa**), ou uma lista de objetos da classe `Funcionario` na classe `Empresa` (**associação unidirecional de Empresa para Funcionario**), ou ambos os casos (**associação bidirecional**).

Repare que no caso de multiplicidade muitos, existem casos em que se faz necessário implementar a associação utilizando uma lista de objetos da outra classe. Este é o caso onde, a partir de uma empresa, desejamos obter os dados dos seus funcionários. Como a empresa não possui apenas um funcionário, não é possível implementar a associação com um objeto simples. Logo, uma lista de funcionários permite o armazenamento dos diversos funcionários da empresa.

#### 4.3.1. Caso unidirecional

Consideremos o caso unidirecional. Se quisermos que o vínculo seja implementado pela classe `Empresa`, então temos uma associação unidirecional de `Empresa` para `Funcionario`. Sua representação UML é apresentada na Figura 4.5.

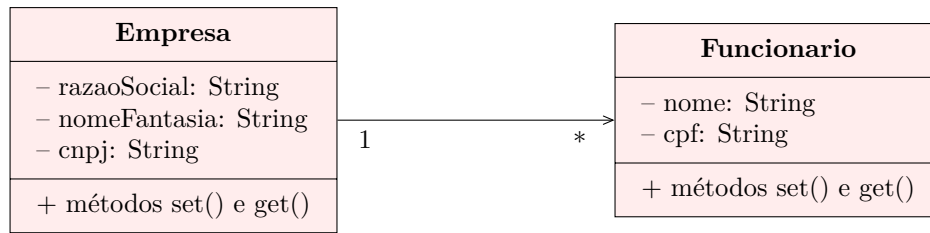


Figura 4.5.: Associação unidirecional de Empresa para Funcionario

Neste caso, precisamos implementar uma lista de objetos de **Funcionario** na classe **Empresa**. Os trechos de código abaixo mostram a implementação de ambas as classes para este caso. Uma boa prática consiste em definir um método específico para adicionar um objeto na lista. Este método é apresentado nas linhas 7 a 9 da classe **Empresa**. Sempre que for necessário vincular um funcionário à empresa, este método pode (ou deve) ser utilizado.

**OBS:** podemos substituir a multiplicidade muitos por **0..\*** ou **1..\***, definindo o mínimo de objetos que a lista deve conter. No caso da multiplicidade **0..\*** a lista pode conter nenhum, um ou muitos objetos vinculados. No caso da multiplicidade **1..\***, a lista deve conter ao menos um objeto vinculado.

```

1  public class Empresa {
2      private String razaoSocial;
3      private String nomeFantasia;
4      private String cnpj;
5      private List<Funcionario> funcionarios;
6
7      public Empresa() {
8          funcionarios = new ArrayList<Funcionario>();
9      }
10
11     public void addFuncionario(Funcionario f) {
12         this.funcionarios.add(f);
13     }
14
15     //demais métodos
16 }
  
```

```

1  public class Funcionario {
2      private String nome;
3      private String cpf;
4
5      public String getNome() {
6          return nome;
7      }
8
9      public void setNome(String nome) {
  
```

```
10     this.nome = nome;
11 }
12
13 //demais métodos
14 }
```

O trecho de código a seguir mostra o uso da associação implementada. São criados distintos objetos das classes `Funcionario` e `Empresa` (linhas 2 a 7). Após isso, os funcionários são atribuídos às respectivas empresas utilizando o método `addFuncionario` (linhas 9 a 11). Finalmente, todos os funcionários da empresa `e1` são apresentados (linhas 13 e 14). Perceba que a lista de funcionários da empresa `e1` é recuperada e, então, seus dados são apresentados em tela.

```
1 public static void main(String[] args) {
2     Funcionario f1 = new Funcionario("José da Silva",
    ↪ "012.541.379-33");
3     Funcionario f2 = new Funcionario("Maria Pereira",
    ↪ "062.411.632-12");
4     Funcionario f3 = new Funcionario("Pedro Ferreira",
    ↪ "178.219.475-25");
5
6     Empresa e1 = new Empresa("Empresa 1 LTDA", "Empresa 1",
    ↪ "123.456.789/0001-01");
7     Empresa e2 = new Empresa("Empresa 2 LTDA", "Empresa 2",
    ↪ "123.456.789/0001-02");
8
9     e1.addFuncionario(f1);
10    e1.addFuncionario(f2);
11    e2.addFuncionario(f3);
12
13    for(Funcionario f : e1.getFuncionarios())
14        System.out.println(f.getNome() + " é funcionário na " +
    ↪ e1.getNomeFantasia());
15 }
```

A mesma análise anterior pode ser feita na associação com multiplicidade muitos. Mantendo uma lista de funcionários para cada empresa, podemos acessar os dados dos funcionários a partir da classe `Empresa`. Porém, não é possível acessar os dados da empresa a partir da classe `Funcionario`. Para modificar isso, podemos inverter a navegabilidade da associação, de modo a manter o vínculo na classe `Funcionario`. Neste caso, o funcionário tem um atributo simples do tipo `Empresa`. Este caso está representado no diagrama da Figura 4.6 e sua implementação segue os conceitos apresentados nas seções anteriores, podendo ser observada nos códigos abaixo.

**OBS:** como a multiplicidade indica que o funcionário pertence a exatamente uma empresa, o vínculo é obrigatório.

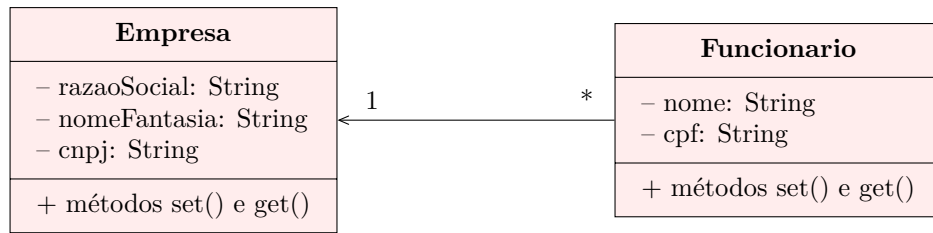


Figura 4.6.: Associação unidirecional de Funcionario para Empresa

```
1 public class Empresa {
2     private String razaoSocial;
3     private String nomeFantasia;
4     private String cnpj;
5
6     public String getRazaoSocial() {
7         return razaoSocial;
8     }
9
10    public void setRazaoSocial(String r) {
11        this.razaoSocial = r;
12    }
13 }
```

```
1 public class Funcionario {
2     private String nome;
3     private String cpf;
4     private Empresa empresa;
5
6     public Empresa getEmpresa() {
7         return empresa;
8     }
9
10    public void setEmpresa(Empresa empresa) {
11        this.empresa = empresa;
12    }
13 }
```

```
1 public static void main(String[] args) {
2     Funcionario f1 = new Funcionario("José da Silva",
3     ↪ "012.541.379-33");
4     Funcionario f2 = new Funcionario("Maria Pereira",
5     ↪ "062.411.632-12");
6     Funcionario f3 = new Funcionario("Pedro Ferreira",
7     ↪ "178.219.475-25");
8 }
```

```

6      Empresa e1 = new Empresa("Empresa 1 LTDA", "Empresa 1",
  ↪    "123.456.789/0001-01");
7      Empresa e2 = new Empresa("Empresa 2 LTDA", "Empresa 2",
  ↪    "123.456.789/0001-02");
8
9      f1.setEmpresa(e1);
10     f2.setEmpresa(e1);
11     f3.setEmpresa(e2);
12
13     System.out.println(f1.getNome() + " é func. na " +
  ↪    f1.getEmpresa().getNomeFantasia());
14     System.out.println(f2.getNome() + " é func. na " +
  ↪    f2.getEmpresa().getNomeFantasia());
15     System.out.println(f3.getNome() + " é func. na " +
  ↪    f3.getEmpresa().getNomeFantasia());
16 }

```

### 4.3.2. Caso bidirecional

Se quisermos que ambas as classes possuam atributos que implementem o vínculo, podemos adotar uma associação bidirecional. Neste caso, existe uma lista de objetos de **Funcionario** na classe **Empresa**, e um objeto da classe **Empresa** na classe **Funcionario**. Com isso, é possível acessar todos os funcionários de uma empresa, bem como a empresa em que um determinado funcionário trabalha. A Figura 4.7 apresenta a representação UML com relacionamento bidirecional entre as classes.

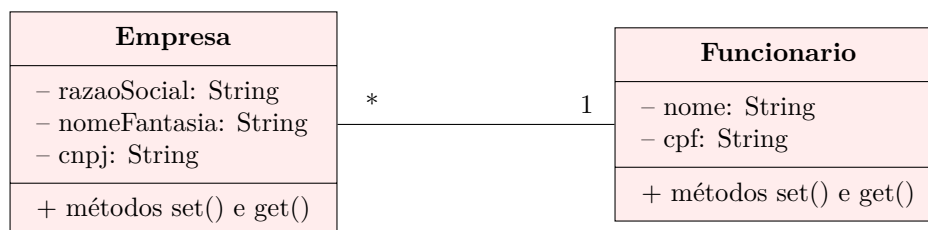


Figura 4.7.: Associação bidirecional entre **Empresa** e **Funcionario**

Os trechos de código a seguir mostram a implementação da estrutura de classes apresentada na Figura 4.7. Repare nos atributos que implementam o vínculo bidimensional. Na classe **Empresa** temos a lista de funcionários (**funcionarios**), enquanto na classe **Funcionario** temos o vínculo com a empresa (**empresa**). Repare ainda que o método **setEmpresa** da classe **Funcionario** também realiza a operação de inclusão do funcionário na lista de funcionários da empresa.

```

1  public class Empresa {
2      private String razaoSocial;
3      private String nomeFantasia;
4      private String cnpj;
5      private List<Funcionario> funcionarios;

```

```
6
7     public void addFuncionario(Funcionario f){
8         this.funcionarios.add(f);
9     }
10 }
```

```
1     public class Funcionario {
2         private String nome;
3         private String cpf;
4         private Empresa empresa;
5
6         public void setEmpresa(Empresa empresa){
7             this.empresa = empresa;
8             empresa.addFuncionario(this);
9         }
10     }
```

O código abaixo mostra o uso da estrutura proposta. A atribuição é feita pela chamada do método `setEmpresa` (linhas 9 a 11). As linhas 13 a 15 mostram que é possível acessar todos os funcionários da empresa (`e1.getFuncionarios()`), bem como acessar os dados da empresa em que um funcionário trabalha (`f.getEmpresa().getNomeFantasia()`).

```
1     public static void main(String[] args) {
2         Funcionario f1 = new Funcionario("José da Silva",
3         ↪ "012.541.379-33");
4         Funcionario f2 = new Funcionario("Maria Pereira",
5         ↪ "062.411.632-12");
6         Funcionario f3 = new Funcionario("Pedro Ferreira",
7         ↪ "178.219.475-25");
8
9         Empresa e1 = new Empresa("Empresa 1 LTDA", "Empresa 1",
10        ↪ "123.456.789/0001-01");
11        Empresa e2 = new Empresa("Empresa 2 LTDA", "Empresa 2",
12        ↪ "123.456.789/0001-02");
13
14        f1.setEmpresa(e1);
15        f2.setEmpresa(e1);
16        f3.setEmpresa(e2);
17
18        for(Funcionario f : e1.getFuncionarios())
19            System.out.println(f.getNome() + " trabalha na " +
20            ↪ f.getEmpresa().getNomeFantasia());
21    }
```

## 4.4. Exemplo – Clube e Sócio

Considere o contexto de um clube e seus sócios, onde um clube possui vários sócios. O diagrama de classes é apresentado pela Figura 4.8. Pela navegabilidade proposta, a classe **Clube** possui uma lista de objetos da classe **Socio**, podendo essa lista ser vazia (multiplicidade 0..\*).

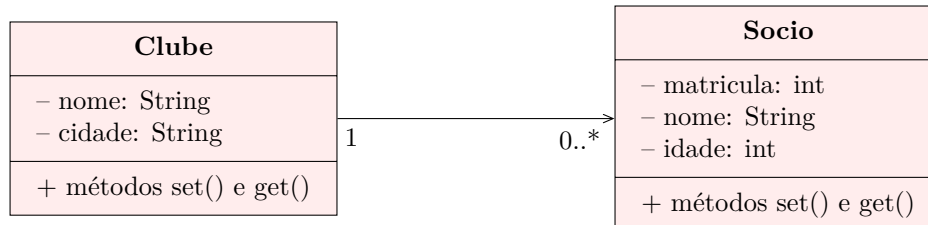


Figura 4.8.: Diagrama de classes para Clube e Socio

Os trechos de código abaixo mostram a implementação das classes **Clube** e **Socio**. Perceba que a lista que implementa a associação encontra-se na classe **Clube** (linha 4), conforme definido na navegabilidade. Esta classe ainda implementa o método `addSocio`, responsável por inserir o sócio recebido como argumento na lista. Além disso, a classe implementa o método `removeSocio`, que recebe a matrícula do sócio a ser removido, busca o objeto desejado na lista e o remove, caso encontrado. Este método devolve verdadeiro, em caso de sucesso, e falso, caso contrário.

```
1 public class Clube {
2     private String nome;
3     private String cidade;
4     private List<Socio> socios;
5
6     public Clube() {
7         socios = new ArrayList<Socio>();
8     }
9
10    public Clube(String nome, String cidade) {
11        this.nome = nome;
12        this.cidade = cidade;
13        this.socios = new ArrayList<Socio>();
14    }
15
16    public void addSocio(Socio socio) {
17        this.socios.add(socio);
18    }
19
20    public boolean removeSocio(int matricula) {
21        for(Socio s : socios) {
22            if(s.getMatricula() == matricula) {
23                this.socios.remove(s);
```

```
24         return true;
25     }
26 }
27     return false;
28 }
29
30 public List<Socio> getSocios() {
31     return socios;
32 }
33
34 public void setSocios(List<Socio> socios) {
35     this.socios = socios;
36 }
37
38 public String getNome() {
39     return nome;
40 }
41
42 public void setNome(String nome) {
43     this.nome = nome;
44 }
45
46 public String getCidade() {
47     return cidade;
48 }
49
50 public void setCidade(String cidade) {
51     this.cidade = cidade;
52 }
53 }
```

```
1 public class Socio {
2     private int matricula;
3     private String nome;
4     private int idade;
5
6     public Socio() {}
7
8     public Socio(int matricula, String nome, int idade) {
9         this.matricula = matricula;
10        this.nome = nome;
11        this.idade = idade;
12    }
13
14    public int getMatricula() {
15        return matricula;
16    }
17 }
```



```
16     }
17
18     public void setMatricula(int matricula) {
19         this.matricula = matricula;
20     }
21
22     public String getNome() {
23         return nome;
24     }
25
26     public void setNome(String nome) {
27         this.nome = nome;
28     }
29
30     public int getIdade() {
31         return idade;
32     }
33
34     public void setIdade(int idade) {
35         this.idade = idade;
36     }
37 }
```

Os trechos de código a seguir mostram o uso da estrutura de classes apresentada acima. O método `insereRegistros` é responsável por criar, vincular e armazenar os registros nas listas definidas na classe `Aplicacao`. O método `removeSocio` seleciona um clube e um sócio, e faz a remoção do sócio da lista de sócios do clube. Em caso de sucesso, o sócio é removido também da lista geral de sócios. O método `mostraRegistros` percorre a lista de clubes, acessando seus dados e sua lista de sócios, para apresentação em tela.

```
1 public class Principal {
2     public static void main(String[] args) {
3         Exemplo exemplo = new Exemplo();
4         exemplo.run();
5     }
6 }
```

```
1 public class Exemplo {
2     private List<Socio> socios = new ArrayList<Socio>();
3     private List<Clube> clubes = new ArrayList<Clube>();
4
5     public void run() {
6         insereRegistros();
7         removeSocio();
8         mostraRegistros();
9     }
}
```

```
10
11 private void insereRegistros() {
12     Socio s1 = new Socio(123456, "Maria da Rosa", 45);
13     Socio s2 = new Socio(654321, "José da Silva", 25);
14     Socio s3 = new Socio(147852, "Ana Lúcia da Silva", 30);
15     Socio s4 = new Socio(369852, "Pedro Ferreira", 28);
16     socios.add(s1);
17     socios.add(s2);
18     socios.add(s3);
19     socios.add(s4);
20
21     Clube c1 = new Clube("Clube ABC", "Ibirama");
22     Clube c2 = new Clube("Clube XYZ", "Blumenau");
23     clubes.add(c1);
24     clubes.add(c2);
25
26     c1.addSocio(s1);
27     c1.addSocio(s2);
28     c2.addSocio(s3);
29     c2.addSocio(s4);
30 }
31
32 public void removeSocio() {
33     Clube c = clubes.get(0);
34     Socio s = socios.get(0);
35
36     if(c.removeSocio(s.getMatricula())) {
37         System.out.println("Sócio [" + s.getMatricula() + "]
↪ removido do clube " + c.getNome());
38         socios.remove(s);
39     } else {
40         System.out.println("Sócio [" + s.getMatricula() + "] não
↪ pertence ao clube " + c.getNome());
41     }
42 }
43
44 private void mostraRegistros() {
45     for(Clube c : clubes) {
46         System.out.println("Sócios do clube " + c.getNome());
47         for(Socio s : c.getSocios()) {
48             System.out.println("- " + s.getNome() + " [" +
↪ s.getMatricula() + "]");
49         }
50         System.out.println("");
51     }
52 }
53 }
```

**SAÍDA:**

Sócios do Clube ABC

- José da Silva [654321]

Sócios do Clube XYZ

- Ana Lúcia da Silva [147852]

- Pedro Ferreira [369852]



## 5. Agregação e composição

A agregação e a composição são tipos específicos de associações, onde uma entidade forma parte de outra. Logo, a implementação de uma agregação ou de uma composição é idêntica à implementação de uma associação simples<sup>1</sup>. Isto é, a diferença entre elas é conceitual.

### 5.1. Relacionamento todo-parte

A agregação e a composição acontecem quando duas entidades possuem um relacionamento todo-parte. Neste relacionamento, uma das entidades forma parte da outra. Por exemplo, uma empresa é composta por departamentos. Os departamentos são partes da empresa. Logo, a entidade **Empresa** é chamada “todo”, enquanto a entidade **Departamento** é chamada “parte”.

Uma instância da classe todo possui uma ou mais instâncias da classe parte. A instância da classe parte complementa as informações da classe todo, de modo que o todo não é completo sem as suas partes. No exemplo da empresa e dos seus departamentos, uma empresa sem departamentos não é completa e os departamentos complementam as informações sobre a empresa. Abaixo são apresentados outros exemplos de entidades que se relacionam como todo-parte.

- Um veículo (todo) é composto por quatro rodas (parte).
- Um computador (todo) possui um teclado, um mouse e um monitor (partes).
- Uma lista de compras (todo) possui uma lista de itens a comprar (parte).
- Uma empresa (todo) é composta por departamentos (parte).
- Um livro (todo) é composto por capítulos (parte).
- Um capítulo do livro (todo) é composto por páginas (parte).

Existem algumas características (apresentadas no formato de perguntas) que auxiliam na identificação de um relacionamento todo-parte:

1. O relacionamento é descrito com uma frase “parte de”?
  - Um **botão** é parte de uma **janela**.
  - A **porta** é parte de um **carro**.

---

<sup>1</sup>Em alguns casos são implementadas regras simples na agregação e na composição, como a obrigatoriedade do vínculo ou a garantia de que uma entidade qualquer não seja componente de duas entidades simultaneamente

---

2. Algumas operações no todo são automaticamente aplicadas a suas partes?
  - Mover a **janela** implica em mover o **botão**.
  - Mover o **carro** implica em mover a **porta**.
3. Alguns valores de atributos são propagados do todo para todos ou algumas de suas partes?
  - A fonte da **janela** é Arial, a fonte do **botão** é Arial.
  - Pintar o carro de **vermelho** implica pintar também a **porta** de vermelho.
4. Existe uma assimetria inerente no relacionamento onde uma classe é subordinada a outra?
  - Uma **botão** É parte de uma **janela**, uma **janela** NÃO É parte de um **botão**.
  - Uma **porta** É parte de um **carro**, um **carro** NÃO É parte de uma **porta**.

Logo, tanto as entidades **janela** e **botão**, quanto as entidades **carro** e **porta** formam relacionamentos todo-parte.

## 5.2. Agregação

Existem duas regras que determinam que o relacionamento todo-parte trata-se de uma agregação:

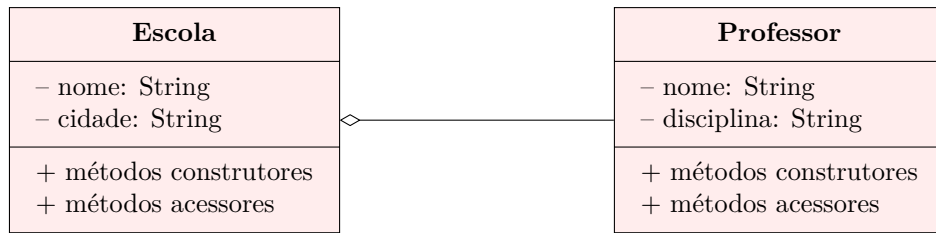
1. Ambas as entidades (todo e parte) podem existir de forma independente à outra.
2. Uma entidade parte pode estar relacionada com mais de uma entidade todo ao mesmo tempo.

**Um exemplo:** uma escola possui vários professores.

- Trata-se de uma agregação, pois um professor pode existir fora do relacionamento com a escola.
- Trata-se de uma agregação, pois um professor pode trabalhar em duas escolas ao mesmo tempo, ou seja, sua entidade se relaciona com dois objetos diferentes do todo.

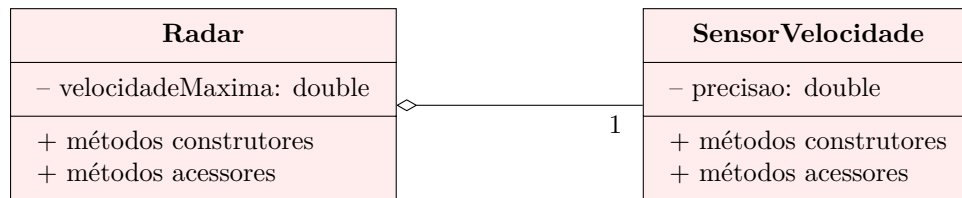
Na UML, representamos uma agregação com um losango não preenchido no lado da entidade que representa o todo. A Figura 5.1 mostra as classes **Professor** e **Escola** e seu relacionamento de agregação. Repare que a entidade todo recebe o losango do relacionamento.

Em geral, o atributo que implementa a agregação fica na entidade todo (apesar de não ser uma regra). Quando a agregação possui multiplicidade 1 no lado parte, ela é implementada por um atributo simples no lado todo. Quando a agregação possui multiplicidade muitos no lado parte, ela é implementada por uma lista de objetos no lado todo.

Figura 5.1.: Relacionamento de agregação entre **Escola** e **Professor**

### 5.2.1. Caso com multiplicidade 1

Um exemplo de agregação com multiplicidade 1 acontece entre as entidades **Radar** e **SensorVelocidade**. Um radar possui um único sensor de velocidade. O sensor de velocidade fica localizado abaixo da pista e, portanto, existe mesmo fora do relacionamento com o radar e pode estar relacionado com dois ou mais radares ao mesmo tempo (regras para definição de uma agregação). A Figura 5.2 mostra o diagrama de classes para este exemplo.

Figura 5.2.: Relacionamento de agregação entre **Radar** e **SensorVelocidade**

Os trechos de código abaixo mostram a implementação das classes apresentadas na Figura 5.2. Na classe **Radar** (entidade todo) o vínculo é implementado pelo atributo **sensor**, que é um objeto da entidade parte.

```

1 public class Radar {
2     private double velocidadeMaxima;
3     private SensorVelocidade sensor;
4
5     public SensorVelocidade getSensor() {
6         return sensor;
7     }
8
9     public void setSensor(SensorVelocidade s){
10         this.sensor = s;
11     }
12 }
  
```

```

1 public class SensorVelocidade {
2     private double precisao;
3
4     public double getPrecisao() {
  
```

```

5     return precisao;
6 }
7
8 public void setPrecisao(double precisao) {
9     this.precisao = precisao;
10 }
11 }

```

### 5.2.2. Caso com multiplicidade muitos

Um exemplo de agregação com multiplicidade muitos ocorre entre uma turma e seus alunos. Os alunos são parte de uma turma, que é a entidade todo. Os alunos existem independente do relacionamento com a turma e um aluno pode pertencer a duas turmas simultaneamente. Logo, temos um relacionamento de agregação. Neste caso, a turma possui vários alunos, definindo a multiplicidade muitos no lado parte. A Figura 5.3 apresenta a estrutura de classes para este exemplo.

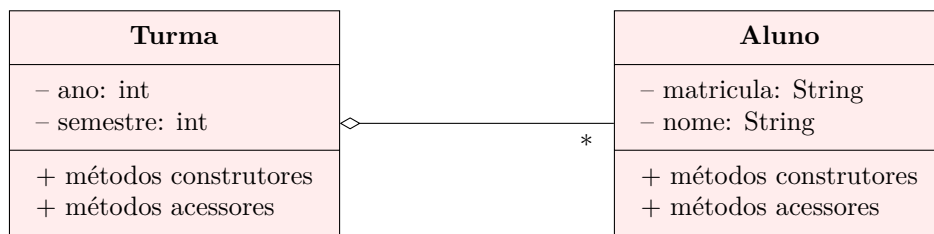


Figura 5.3.: Relacionamento de agregação entre Turma e Aluno

Os códigos a seguir mostram a implementação das classes apresentadas na Figura 5.3. Repare que a agregação é implementada por uma lista na entidade todo. Neste sentido, a classe `Turma` possui uma lista de objetos da classe `Aluno`. Assim como na associação simples, uma boa prática consiste em definir um método `addAluno` para realizar o vínculo entre as duas entidades. Da mesma forma, podem ser implementados métodos para alteração e remoção de alunos (ou outros métodos para manipulação dessas entidades).

```

1 public class Turma {
2     private int ano;
3     private int semestre;
4     private List<Aluno> alunos = new ArrayList<Aluno>();
5
6     public void addAluno(Aluno a) {
7         this.alunos.add(a);
8     }
9
10    public List<Aluno> getAlunos() {
11        return alunos;
12    }
13 }

```



```
14     public void setAlunos(List<Aluno> alunos){
15         this.alunos = alunos;
16     }
17 }
```

```
1  public class Aluno {
2      private String matricula;
3      private String nome;
4
5      public String getMatricula() {
6          return matricula;
7      }
8
9      public void setMatricula(String mat) {
10         this.matricula = mat;
11     }
12 }
```

### 5.3. Composição

Existem duas regras que determinam que o relacionamento todo-parte trata-se de uma composição:

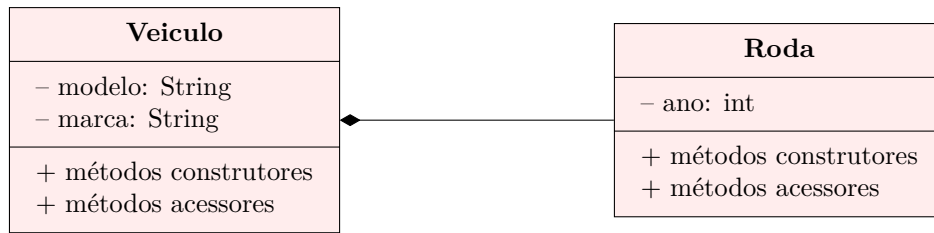
1. A entidade parte só existe em função do relacionamento que possui com a entidade todo. Caso a entidade todo seja destruída, suas partes também são destruídas.
2. Uma entidade parte pode estar relacionada com apenas uma entidade todo simultaneamente.

**Um exemplo:** um veículo possui quatro rodas.

- Trata-se de uma composição, pois uma roda não existe sem estar vinculada ao veículo.
- Trata-se de uma composição, pois uma roda não pode estar vinculada a dois veículos ao mesmo tempo, sua entidade se relaciona a apenas um objeto todo.

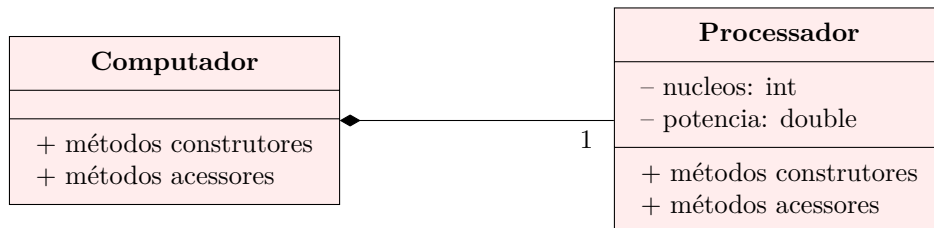
Na UML, representamos uma composição com um losango preenchido no lado da entidade que representa o todo. A Figura 5.4 mostra as classes **Veiculo** e **Roda** e seu relacionamento de composição. Repare que a entidade todo recebe o losango do relacionamento.

Da mesma forma como ocorre com a agregação, um relacionamento de composição com multiplicidade 1 na entidade parte é implementado com um atributo simples na classe todo. Um relacionamento de composição com multiplicidade muitos na entidade parte é implementado com uma lista de objetos na classe todo. Diferente da agregação, um objeto da classe parte está associado a apenas um objeto da classe todo (conceito de composição). Logo, a multiplicidade na parte todo é, obrigatoriamente, 1.

Figura 5.4.: Relacionamento de agregação entre **Veiculo** e **Roda**

### 5.3.1. Caso com multiplicidade 1

Um exemplo de composição com multiplicidade 1 acontece entre as entidades **Computador** e **Processador**. Um computador possui um único processador. O processador fica localizado dentro do computador e, portanto, não existe fora do relacionamento com o computador e não pode estar relacionado com dois ou mais computadores ao mesmo tempo (regras para definição de uma composição). A Figura 5.5 mostra o diagrama de classes para este exemplo.

Figura 5.5.: Relacionamento de composição entre **Computador** e **Processador**

Os trechos de código abaixo mostram a implementação das classes apresentadas na Figura 5.5. Na classe **Computador** (entidade todo) o vínculo é implementado pelo atributo **processador**, que é um objeto da entidade parte. Uma boa prática (mas não uma regra) consiste em não implementar o método **setProcessador**, mas sim um método que receba os atributos do processador e instancie um novo objeto para vínculo (linhas 4 a 8). Com isso, evita-se que uma mesma instância da entidade parte seja vinculada a duas ou mais instâncias da entidade todo. No entanto, com isso não se permite que o objeto parte troque seu vínculo para outro objeto todo. Caso seja preciso, deve-se implementar os métodos acessores necessários.

```

1  public class Computador {
2      private Processador processador;
3
4      public void addProcessador(int nucleos, double potencia) {
5          processador = new Processador();
6          processador.setNucleos(nucleos);
7          processador.setPotencia(potencia);
8      }
9
10     public void removeProcessador() {
11         this.processador = null;
  
```

```

12     }
13 }

1 public class Processador {
2     private int nucleos;
3     private double potencia;
4
5     public int getNucleos() {
6         return nucleos;
7     }
8
9     public void setNucleos(int nucleos) {
10        this.nucleos = nucleos;
11    }
12
13    public double getPotencia() {
14        return potencia;
15    }
16
17    public void setPotencia(double potencia) {
18        this.potencia = potencia;
19    }
20 }

```

### 5.3.2. Caso com multiplicidade muitos

Um exemplo de composição com multiplicidade muitos acontece entre as entidades **Livro** e **Capítulo**. Um livro é composto por vários capítulos. O capítulo é parte de um livro e, portanto, não existe fora do relacionamento. Além disso, um capítulo não deve fazer parte de dois livros ao mesmo tempo. Logo, trata-se de uma composição. A Figura 5.6 mostra o diagrama de classes para este exemplo.

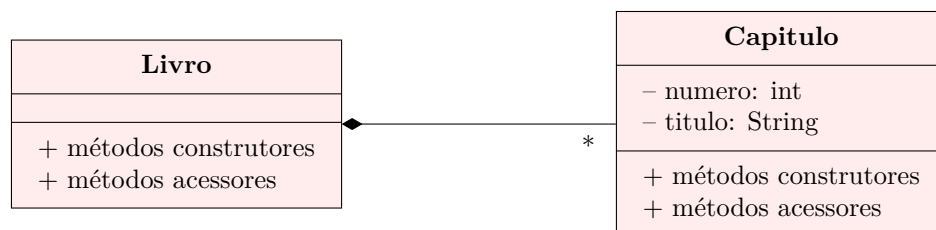


Figura 5.6.: Relacionamento de composição entre Livro e Capítulo

Os trechos de código abaixo mostram a implementação da composição apresentada na Figura 5.6. A composição é definida pela lista de capítulos (atributo **caps**) da classe **Livro**. Novamente, as operações de criação e vínculo dos capítulos ficam a cargo da classe **Livro** (linhas 4 a 10 da classe **Livro**).

```
1 public class Livro {
2     private List<Capitulo> caps = new ArrayList<Capitulo>();
3
4     public void addCapitulo(
5         int numero, String titulo) {
6         Capitulo c = new Capitulo();
7         c.setNumero(numero);
8         c.setTitulo(titulo);
9         caps.add(c);
10    }
11
12    //demais métodos
13 }
```

```
1 public class Capitulo {
2     private int numero;
3     private String titulo;
4
5     public int getNumero() {
6         return numero;
7     }
8
9     public void setNumero(int numero) {
10        this.numero = numero;
11    }
12
13    //demais métodos
14 }
```

## 6. Dependência

Uma classe **A** depende de uma classe **B** quando, no momento da compilação da classe **A**, o código da classe **B** também é compilado. Ou seja, para que a classe **A** funcione, é preciso existir (e funcionar) a classe **B**. Logo, classes que possuem entre si quaisquer relacionamentos (associação, agregação, composição, especialização) possuem uma dependência.

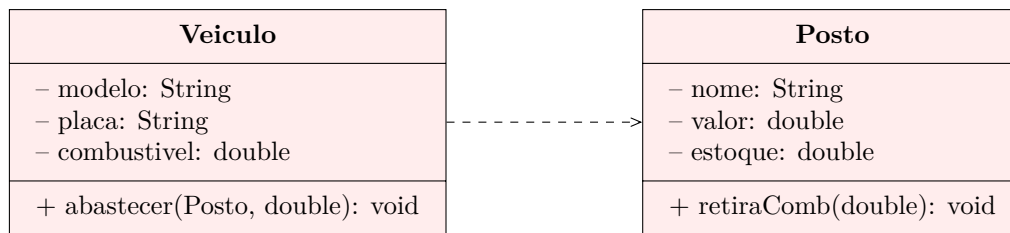
A dependência é dada pela navegabilidade do relacionamento. Por exemplo, em uma associação a classe que possui um objeto da outra, possui uma dependência com a mesma. Caso a classe independente seja apagada, a classe dependente apresentará erro (ou seja, a classe que implementa o vínculo).

Na orientação a objetos, relacionamentos de diferentes naturezas são representados com diferentes tipos (associação, agregação, composição, etc.). Para os casos onde o relacionamento não se encaixa nos tipos predefinidos, o relacionamento é chamado de dependência. Logo, ele pode ser visto como um relacionamento de utilização, onde os objetos não possuem um vínculo semântico, mas um deles (dependente) faz uso dos serviços do outro (independente). Em geral, uma dependência ocorre quando um **objeto da outra classe é utilizado como parâmetro ou retorno de um método, ou quando é utilizado internamente pela classe**. As seções a seguir abordam os diferentes casos de dependência através de exemplos.

### 6.1. Dependência no argumento de um método

Considere as entidades **Veiculo** e **Posto**. Não existe nenhum relacionamento semântico entre estas classes, isto é, elas não são associadas, nem possuem agregação, composição ou herança. No entanto, a classe **Veiculo** deve implementar um método chamado **abastecer**, no qual recebe combustível (incrementa seu atributo **combustivel**) de um posto de gasolina (classe **Posto**), que transfere combustível para o referido veículo (decrementa seu atributo **estoque**). Perceba que a entidade **Veiculo** usa um serviço fornecido pelo objeto da classe **Posto**, mas não possui um relacionamento de associação com esta classe. Logo, o relacionamento consiste em uma dependência.

Podemos implementar esta dependência no argumento do método **abastecer**, onde o veículo recebe como parâmetro o posto de combustível no qual está abastecendo, além da quantidade desejada de combustível, para então utilizar o serviço de retirada de combustível (implementado pelo método **retiraComb** da classe **Posto**). Este cenário é descrito pelo diagrama de classes apresentado na Figura 6.1 e seu código é apresentado na sequência. Observe que a dependência é representada por uma linha tracejada com ponta de seta para a classe independente.

Figura 6.1.: Relacionamento de dependência entre **Veiculo** e **Posto**

Perceba que o método `abastecer` utiliza um objeto da classe **Posto** na sua implementação, recebendo-o como argumento. Após isso, a entidade **Veiculo** utiliza o serviço implementado pelo método `retiraComb` da classe **Posto**, capturando seu retorno para o incremento do atributo `combustivel`. A classe **Posto**, por sua vez, fornece o método `retiraComb` que, com base na quantidade recebida como argumento, verifica a possibilidade de abastecimento, decrementando o combustível do seu estoque, em caso positivo. Ao final, o resultado da retirada é devolvido (verdadeiro ou falso).

```
1 public class Veiculo {
2     private String modelo;
3     private String placa;
4     private double combustivel;
5
6     public void abastecer(Posto posto, double qtd) {
7         if(posto.retiraComb(qtd))
8             this.combustivel += qtd;
9     }
10
11     //demais métodos
12 }
```

```
1 public class Posto{
2     private String nome;
3     private double valor;
4     private double estoque;
5
6     public boolean retiraComb(double qtd) {
7         if(qtd <= estoque) {
8             estoque -= qtd;
9             return true;
10        }
11        return false;
12    }
13
14    //demais métodos
15 }
```

O relacionamento existente entre as classes **Veiculo** e **Posto** é de utilização, onde a

primeira classe utiliza a segunda. Não existe um vínculo entre as entidades, o que caracteriza a dependência. A classe `Veiculo` é dependente da classe `Posto`. Observe que a exclusão da classe `Posto` implica em erro na classe `Veiculo` (pois a última **depende** da primeira).

## 6.2. Dependência no retorno de um método

Um segundo tipo de dependência ocorre quando o relacionamento de utilização se dá no retorno do método. Um exemplo disso pode ser observado agregando mais uma característica no exemplo anterior. Se quisermos implementar um método que calcule o desempenho do veículo em termos de emissão de poluentes, emissão de ruído e consumo de combustível. Podemos definir uma classe que agregue as diferentes métricas de desempenho. O código abaixo apresenta a implementação da classe `Metrica`, que reúne as três medidas supracitadas.

```
1 public class Metrica {
2     private double emissao;
3     private double ruido;
4     private double consumo;
5
6     public double getEmissao() {
7         return emissao;
8     }
9
10    public void setEmissao(double emissao) {
11        this.emissao = emissao;
12    }
13
14    //demais métodos
15 }
```

Diante disso, queremos implementar um método na classe `Veiculo` que calcule as medidas de desempenho em função da aceleração. Logo, o método `avaliar` recebe como argumento a aceleração praticada e determina os valores de emissão de poluentes, emissão de ruído e consumo de combustível, reunindo-os em um objeto da classe `Metrica`, que é devolvido como retorno do método. A Figura 6.2 apresenta o diagrama de classes atualizado. Com isso, a classe `Veiculo` depende da classe `Metrica` pois implementa um método cujo retorno é um objeto desta classe.

O trecho de código abaixo mostra a implementação atualizada da classe `Veiculo`, incluindo o método `avaliar`. De acordo com o valor de aceleração, este método define diferentes valores para as medidas reunidas na classe `Metrica`. Ao final, o método devolve um objeto `Metrica` com os valores definidos.

```
1 public class Veiculo {
2     private String modelo;
```

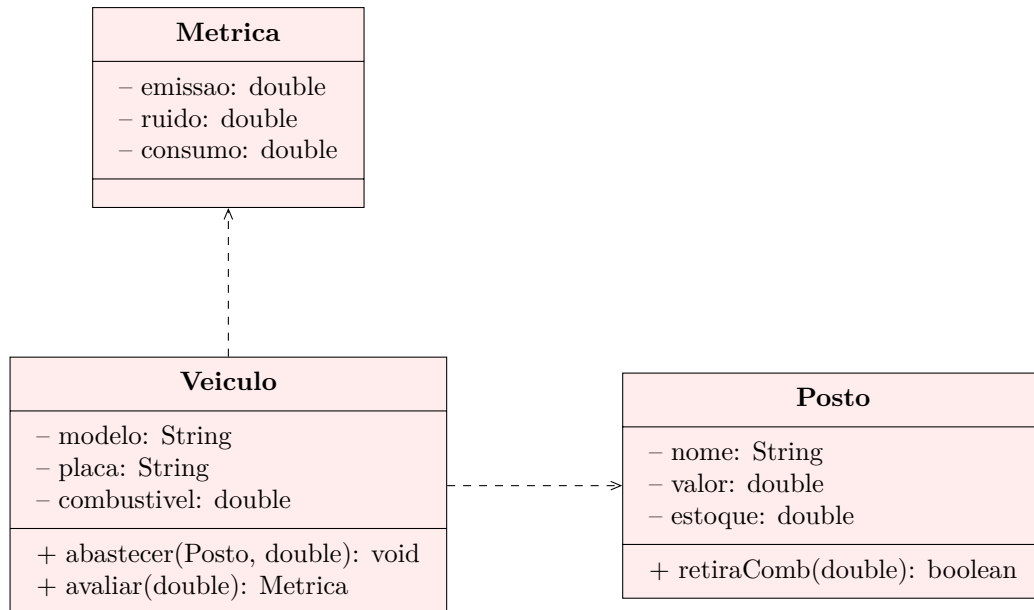


Figura 6.2.: Relacionamento de dependência entre **Veiculo**, **Posto** e **Metrica**

```

3  private String placa;
4  private double combustivel;
5
6  public void abastecer(Posto posto, double qtd) {
7      if(posto.retiraComb(qtd))
8          this.combustivel += qtd;
9  }
10
11 public Metrica avaliar(double aceleracao) {
12     Metrica m = new Metrica();
13     if(aceleracao <= 10) {
14         m.setConsumo(12);
15         m.setRuido(41);
16         m.setEmissao(340);
17     } else {
18         m.setConsumo(6);
19         m.setRuido(70);
20         m.setEmissao(510);
21     }
22     return m;
23 }
24
25 //demais métodos
26 }
  
```



### 6.3. Dependência por uso interno

Um terceiro tipo de dependência ocorre quando uma classe utiliza outra como tipo de atributo, mas não caracterizando como um vínculo de associação, agregação ou composição. Por exemplo, uma classe `Aplicacao` que mantém uma lista de objetos da classe `Aluno`, na qual armazena os registros recuperados de um banco de dados para apresentação em tela. Neste caso, as entidades envolvidas na relação não possuem vínculo semântico, mas a exclusão da classe `Aluno` implica em erro na classe `Aplicacao`. Logo, ambas as classes possuem uma dependência. A Figura 6.3 apresenta o diagrama de classes para este exemplo, mostrando que a classe `Aplicacao` depende da classe `Aluno`. Repare que, neste caso, o atributo que define a dependência pode ser representado no diagrama de classes, evidenciando o tipo de dependência existente entre as classes.

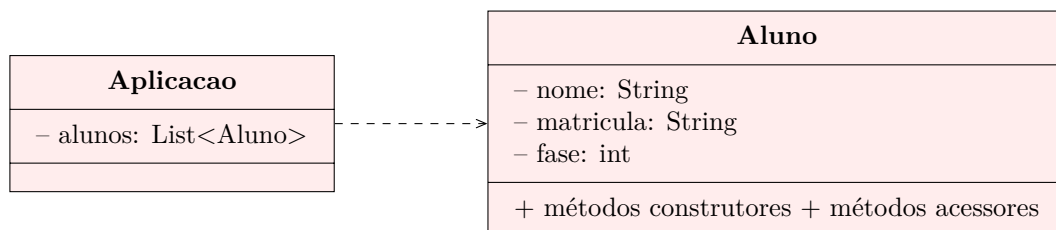


Figura 6.3.: Relacionamento de dependência entre `Aplicacao` e `Aluno`

Os trechos de código abaixo mostram a implementação das classes `Aplicacao` e `Aluno`. O atributo `alunos` define o relacionamento de dependência, conforme apresentado pela Figura 6.3.

```
1 public class Aplicacao {
2     private List<Aluno> alunos;
3
4     public Aplicacao() {
5         this.alunos = new ArrayList<Aluno>();
6     }
7
8     //demais métodos
9 }
```

```
1 public class Aluno {
2     private String nome;
3     private String matricula;
4     private int fase;
5
6     public void setNome(String nome) {
7         this.nome = nome;
8     }
9
10    public String getNome() {
11        return this.nome;
12    }
13 }
```

```

12     }
13
14     //demais métodos
15 }

```

Este tipo de dependência também acontece quando uma classe define um objeto da outra classe dentro de um método. Por exemplo, se uma classe **A** deseja usar as operações fornecidas por uma classe **B** na implementação de um método. Neste caso, o objeto da classe **B** é instanciado dentro do referido método, caracterizando uma dependência entre as classes envolvidas.

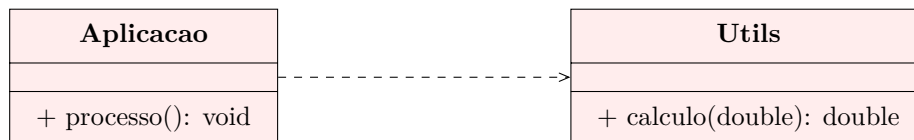


Figura 6.4.: Relacionamento de dependência entre **Aplicacao** e **Utils**

Um exemplo deste tipo de dependência é apresentado pela Figura 6.4. A classe **Utils** é uma classe utilitária, que fornece métodos para outras classes utilizarem. A classe **Aplicacao** utiliza a classe **Utils**, que fornece o método **calculo**. Sua utilização é feita dentro do método **processo**. A implementação destas classes é apresentada nos trechos de código abaixo.

```

1  public class Aplicacao {
2
3      public void processo() {
4          double valor = 40;
5          Utils utils = new Utils();
6          double resultado = utils.calculo(valor);
7          resultado += 10;
8          System.out.println(resultado);
9      }
10
11     //demais métodos
12 }

```

```

1  public class Utils {
2
3      public double calculo(double valor) {
4          return (valor + 40) * (valor / 5);
5      }
6
7      //demais métodos
8  }

```

Repare que o objeto é criado e instanciado dentro do método **processo** da classe **Aplicacao**. Logo, trata-se de uma dependência por uso interno.

**OBS:** o termo *dependência por atributo* não é amplamente utilizado, podendo este tipo de dependência ser encontrado com outros nomes na literatura.



## 7. Herança e polimorfismo

### 7.1. Herança

A herança permite definir elementos específicos, que incorporam a estrutura (atributos) e o comportamento (operações) de elementos mais gerais. Neste sentido, a classe específica herda a estrutura e o comportamento da classe geral, definindo uma hierarquia entre elas. Por conta disso, a herança é também chamada de especialização ou generalização. Com o uso da herança é possível reduzir a escrita de código, diminuindo a redundância e agregando flexibilidade e manutenibilidade ao projeto. Na UML, a herança é representada por uma linha sólida contendo um triângulo no lado da classe mais geral, conforme exemplificado na Figura 7.1.

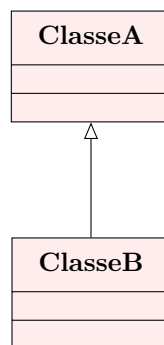


Figura 7.1.: Representação UML de uma herança

Consideremos o contexto de uma empresa a qual possui funcionários e gerentes. Estas entidades são modeladas pelas classes **Funcionario** e **Gerente**. Todo o funcionário da empresa possui uma matrícula e um salário, inclusive os gerentes. Porém, cada gerente possui um número de subordinados e uma senha para acesso ao sistema. Os trechos de código a seguir mostram a implementação das classes **Funcionario** e **Gerente**.

```
1 public class Funcionario {
2     private String matricula;
3     private double salario;
4
5     //...
6 }
```

```
1 public class Gerente {
2     private String matricula;
```

```
3     private double salario;  
4     private int subordinados;  
5     private int senha;  
6  
7     //...  
8 }
```

Podemos perceber que há redundância de código, pois ambas as classes definem os atributos `matricula` e `salario` (linhas 2 e 3 nas duas classes). E se forem incluídos outros tipos de funcionários (secretária, diretor, presidente)? Neste caso, o código deverá ser replicado nas novas classes. E se, após criados os funcionários, seus atributos tiverem que ser alterados? Então cada classe deverá ser alterada. Ou seja, a estrutura adotada não é flexível nem manutenível, o que leva a um software de baixa qualidade.

A solução para este problema está na utilização de herança, de modo que uma classe geral define a estrutura e o comportamento básicos de um funcionário, enquanto classes específicas herdam estas características e adicionam o necessário para cada tipo de funcionário. Neste sentido, a classe `Funcionario` passa a ser a classe geral, definindo os atributos `matricula` e `salario` enquanto a classe `Gerente` herda estes atributos e adiciona os atributos específicos de um gerente, isto é, `subordinados` e `senha`. A estrutura das classes utilizando herança é apresentada pela Figura 7.2.

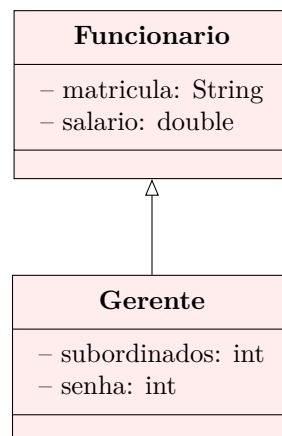


Figura 7.2.: Relacionamento de herança entre `Funcionario` e `Gerente`

Uma herança define um relacionamento do tipo **É UM**. Neste caso, um gerente **É UM** funcionário (ou então, um gerente **É UM TIPO DE** funcionário). Repare que o contrário não é verdadeiro, isto é, um funcionário **NÃO É UM** gerente (não necessariamente). Quando lemos o relacionamento da classe mais geral para a mais específica, chamamos de especialização: *um gerente é uma especialização de um funcionário*. Quando lemos o relacionamento da classe mais específica para a mais geral, chamamos de generalização: *um funcionário é uma generalização de um gerente*. A classe geral recebe o nome de **super-classe** ou **classe-mãe** (neste caso, a classe `Funcionario`), enquanto a classe específica recebe o nome de **subclasse** ou **classe-filha** (neste caso, a classe `Gerente`).

Os trechos de código abaixo mostram a implementação das classes usando herança, conforme proposto na Figura 7.2. A implementação de uma herança é feita utilizando a palavra reservada `extends`, onde se define que a classe específica estende a classe geral.

Neste caso, `Gerente` estende um `Funcionario`, pois herda suas características e inclui características adicionais.

```
1 public class Funcionario {
2     private String matricula;
3     private double salario;
4
5     //...
6 }
```

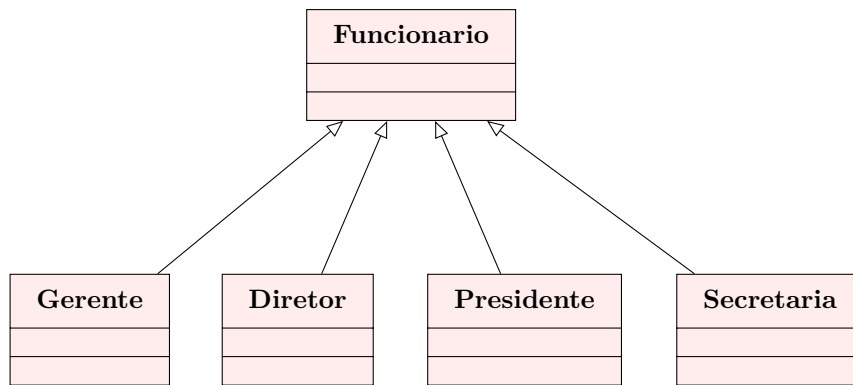
```
1 public class Gerente extends Funcionario {
2     private int subordinados;
3     private int senha;
4
5     //...
6 }
```

O trecho de código a seguir mostra o uso da estrutura de classes criada. Um objeto da classe `Gerente` tem acesso a tudo que for público na classe `Funcionario`. Porém, um objeto da classe `Funcionario` não herda nada da classe `Gerente`. Perceba que o objeto `g` acessa os métodos acessores dos atributos `matricula` e `salario` (linhas 3 e 4), os quais são implementados pela classe `Funcionario` e herdados no relacionamento de herança.

**OBS:** Apesar da classe `Gerente` herdar os atributos de `Funcionario`, ela não pode acessá-los diretamente, pois são privados (o acesso é feito pelo método acessor correspondente). Uma solução para isso seria utilizar outro modificador de acesso: o `protected` – protegido.

```
1 public static void main(String[] args) {
2     Gerente g = new Gerente();
3     g.setMatricula("123456");
4     g.setSalario(4500);
5     g.setSubordinados(10);
6     g.setSenha(1234);
7 }
```

A estrutura proposta (e apresentada na Figura 7.2) pode crescer, de modo a incluir diferentes tipos de funcionários. Neste caso, cada novo tipo de funcionário é implementado por uma nova classe que herda as características da classe `Funcionario`. Com isso, caso no futuro o atributo matrícula seja alterado para um número de registro do tipo `int`, a alteração afetará somente a classe `Funcionario`. Todos os demais tipos de funcionários passam a herdar as novas características, sem necessidade de alteração do código em diferentes pontos do sistema (isso significa flexibilidade e manutenibilidade). A Figura 7.3 mostra este cenário, com diferentes tipos de funcionário se relacionando com a classe `Funcionario` por meio de herança.

Figura 7.3.: Estrutura de classes herdando as características de **Funcionario**

## 7.2. Sobrescrita de método

Consideremos a situação onde todos os funcionários da empresa têm direito a uma gratificação de natal, que consiste em 50% do seu salário. Logo, podemos implementar o método que determina a gratificação na classe **Funcionario**, pois todos os funcionários têm direito a ela. Os códigos a seguir mostram a implementação deste método.

```
1 public class Funcionario {
2     private String matricula;
3     private double salario;
4
5     public double gratificacao() {
6         return this.salario * 0.5;
7     }
8
9     //...
10 }
```

```
1 public class Gerente extends Funcionario {
2     private int subordinados;
3     private int senha;
4
5     //...
6 }
```

Repare que o método **gratificacao**, sendo público, também é herdado pelas subclasses de **Funcionario**. Com isso, a classe **Gerente** herda este método e um objeto desta classe é capaz de chamá-lo. Esta situação é apresentada no código a seguir, que imprimirá em tela o valor de **2250.0**, que consiste em 50% do salário do gerente criado.

```
1 public static void main(String[] args) {
2     Gerente g = new Gerente();
3     g.setMatricula("123456");
```



```
4     g.setSalario(4500);
5     g.setSubordinados(10);
6     g.setSenha(1234);
7     System.out.println(g.gratificacao()); //Imprimirá 2250.0
8 }
```

Consideremos agora que a empresa decida conceder aos gerentes da empresa uma gratificação diferenciada, dado o fato de ser um cargo de confiança. Agora, a gratificação dos gerentes passa a ser 75% do seu salário. Como resolver este problema?

**Opção 1:** criar um segundo método chamado `gratificacaoGerente`.

- **Problema 1:** a classe `Gerente` possuirá dois métodos de gratificação, deixando-a confusa e permitindo a chamada do método errado.
- **Problema 2:** caso a gratificação do diretor seja diferente, um terceiro método deve ser criado, e assim sucessivamente.

**Opção 2 (melhor opção):** reescrever o método `gratificacao` na classe `Gerente`.

- Isso é sobrescrita de métodos!

Os trechos de código abaixo mostram a implementação da sobrescrita do método `gratificacao`. Agora, a classe `Funcionario` possui uma implementação do referido método, enquanto a classe `Gerente` possui uma implementação diferente. Todo o objeto da classe `Funcionario` executará o método `gratificacao` da sua classe e todo o objeto da classe `Gerente` executará o método `gratificacao` da sua classe.

**OBS:** uma boa prática é utilizar a anotação `@Override`, indicando que o método foi sobrescrito da sua superclasse.

```
1 public class Funcionario {
2     private String matricula;
3     private double salario;
4
5     public double gratificacao() {
6         return this.salario * 0.5;
7     }
8
9     //...
10 }
```

```
1 public class Gerente extends Funcionario {
2     private int subordinados;
3     private int senha;
4
5     @Override
6     public double gratificacao() {
7         return this.getSalario() * 0.75;
8     }
9 }
```

```
9
10 //...
11 }
```

O trecho de código abaixo mostra o uso da estrutura de classes criada acima, onde um objeto **Gerente** e um objeto **Funcionario** são criados. Ao chamar o método **gratificacao** do gerente, 75% do salário é impresso em tela. Ao fazer o mesmo com o funcionário, 50% do salário é impresso em tela. Ou seja, cada objeto executa o método implementado na sua classe.

**OBS:** caso o objeto não tivesse implementação do método na sua classe, ele executaria a implementação herdada, isto é, a implementação definida na classe **Funcionario**.

```
1 public static void main(String [] args) {
2     Gerente g = new Gerente();
3     g.setSalario(1000);
4
5     Funcionario f = new Funcionario();
6     f.setSalario(1000);
7
8     System.out.println(g.gratificacao()); //imprimirá 750.0
9     System.out.println(f.gratificacao()); //imprimirá 500.0
10 }
```

Neste ponto, se quisermos incluir um novo tipo de funcionário, basta criarmos a classe para o novo tipo e herdarmos as características de **Funcionario**. Isso faz com que o novo tipo de funcionário herde os atributos e também o método **gratificacao**. Caso queiramos que a gratificação para este novo tipo de funcionário seja diferenciada, basta sobrecrevermos o método **gratificacao** (novamente, flexibilidade e manutenibilidade).

Consideremos agora um novo cenário: a gratificação concedida aos gerentes deve ser igual àquela concedida aos funcionários, com um acréscimo de R\$500,00. Uma solução para este novo cenário consiste em implementar a gratificação de 50% do salário, mais o adicional de R\$500,00 (ou seja, copiar a implementação da classe funcionário, somando ainda o valor adicional). O trecho de código a seguir mostra a implementação do referido método.

```
1 @Override
2 public double gratificacao() {
3     return this.getSalario() * 0.5 + 500;
4 }
```

No entanto, se a gratificação dos funcionários for alterada (para 60%, por exemplo), o método da classe **Gerente** também deverá ser alterado. Ou seja, uma mudança que exige alteração do código em dois pontos. O mesmo pode ocorrer ainda com diferentes tipos de funcionário.

Uma solução mais adequada consiste em chamar o método **gratificacao** da superclasse (**Funcionario**) e acrescentar os R\$500,00. O acesso à superclasse é feito através da

cláusula `super`, que devolve a instância da superclasse da herança, permitindo a execução do método implementado nela. Neste caso, permitindo a chamada do método `gratificacao` da classe `Funcionario`. O trecho de código abaixo mostra este exemplo.

```
1  @Override
2  public double gratificacao() {
3      return super.gratificacao() + 500;
4  }
```

Com isso, caso a gratificação dos funcionários seja alterada, apenas o código da classe `Funcionario` terá de ser alterado. Sempre que a subclasse deve fazer “**algo a mais**” em relação à implementação da superclasse, esta técnica deve ser aplicada, garantindo (mais uma vez) a flexibilidade e a manutenibilidade.

### 7.3. Polimorfismo

Na herança desenvolvida nas seções anteriores, um gerente **É UM** funcionário. Se uma emissora de televisão fizer um convite para que um dos funcionários da empresa conceda uma entrevista, um gerente poderá fazê-lo, pois o gerente é um funcionário. Uma variável do tipo `Funcionario` armazena uma referência a um `Funcionario`. Logo, ela pode armazenar uma referência a um `Gerente`, pois este é um `Funcionario`. O trecho de código abaixo exemplifica esta possibilidade.

```
1  Gerente g1 = new Gerente();
2  Funcionario f1 = g1;
3  Funcionario f2 = new Gerente();
```

**Polimorfismo** é, portanto, a capacidade de um objeto poder ser referenciado de várias formas. No exemplo anterior, o objeto `f1` pode armazenar uma referência a um objeto da classe `Funcionario` ou uma referência a um objeto da classe `Gerente`. Se tivéssemos mais classes estendendo `Funcionario`, ele poderia armazenar uma referência a um objeto de qualquer uma dessas classes (várias formas – polimorfismo).

Considerando um objeto do tipo `Funcionario` armazenando uma referência a um objeto da classe `Gerente`. Ao chamar o método `gratificacao` (sobrescrito na classe `Gerente`), qual das implementações será executada? Considere que `Funcionario` implementa uma gratificação de 50% e `Gerente` implementa uma gratificação de 75%.

```
1  Funcionario f = new Gerente();
2  f.setSalario(1000);
3  System.out.println(f.gratificacao());
```

**Resposta:** a decisão sobre qual método executar é feito em tempo de execução. O Java verifica qual a classe do objeto que está sendo referenciado dentro da variável e executa o respectivo método. Neste caso, executará o método implementado na classe `Gerente` (que é a classe da referência armazenada em `f`), imprimindo o valor de `750.0`.

O polimorfismo é especialmente útil se quisermos definir um método genérico para todos os funcionários, independente do seu tipo (gerente, diretor, secretária, etc.). A classe apresentada no código abaixo tem a responsabilidade de controlar o total de gratificações concedidas. O método `registro` recebe um funcionário e computa a gratificação do mesmo, armazenando a soma total no atributo `totalGratificacoes`.

```
1 public class ControleGratificacoes {
2     private double totalGratificacoes = 0;
3
4     public void registro(Funcionario f) {
5         this.totalGratificacoes += f.gratificacao();
6     }
7
8     public double getTotalGratificacoes() {
9         return this.totalGratificacoes;
10    }
11 }
```

O método `registro` recebe uma referência a um `Funcionario`, chamando seu método `gratificacao`. Logo, ele pode receber referências a qualquer classe que estende `Funcionario`, verificando a referência recebida e chamando o método correto. Ou seja, não importa se o objeto recebido seja um gerente ou uma secretária, o método funcionará de qualquer forma. Mais do que isso, se incluído um novo tipo de funcionário, o método `registro` não precisa ser alterado, pois é genérico para qualquer tipo de funcionário (desde que, claro, estenda a classe `Funcionario`).

## 7.4. Exemplo – Veículos

Considere duas entidades: carro e moto. Um carro possui uma marca, um modelo, uma cor, um valor e um número de portas. Uma moto possui uma marca, um modelo, uma cor, um valor e uma quantidade de cilindradas. Como as classes possuem replicação de código, podemos definir uma classe geral e estendê-la nas classes `Carro` e `Moto`.

Além dos atributos, estas duas entidades possuem em comum um método para cálculo do seu imposto, que corresponde a 2% do seu valor. Especificamente para carros, é acrescido R\$ 800,00 ao seu imposto.

### 7.4.1. Estrutura das classes

A Figura 7.4 mostra a estrutura das classes envolvidas no exemplo. A classe `Veiculo` define as características gerais de um veículo (marca, modelo cor e valor), bem como o método para cálculo do imposto. As classes `Carro` e `Moto` herdam os atributos e métodos da classe `Veiculo`, adicionando suas particularidades (quantidade de portas e cilindradas), assim como sobrescrevendo o método para cálculo do imposto, no caso dos carros.

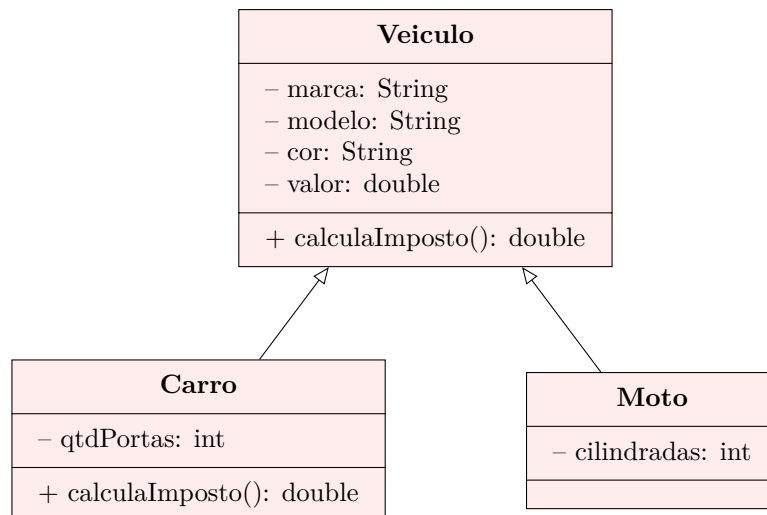


Figura 7.4.: Estrutura de classes para o exemplo dos veículos

### 7.4.2. Implementação das entidades

#### Classe Veiculo

O código abaixo mostra a implementação da classe **Veiculo**, que desempenha o papel da superclasse no exemplo. Esta classe define os atributos comuns de todos os veículos (linhas 2 a 5) e a implementação do método comum a todos os veículos (**calculaImposto**), o qual define o valor do imposto como 2% do valor do veículo.

```
1 public class Veiculo {
2     private String marca;
3     private String modelo;
4     private String cor;
5     private double valor;
6
7     public double calculaImposto() {
8         return this.valor * 0.02;
9     }
10
11     public Veiculo() {}
12
13     public Veiculo(String marca, String modelo, String cor, double
14     ↪ valor) {
15         this.marca = marca;
16         this.modelo = modelo;
17         this.cor = cor;
18         this.valor = valor;
19     }
20     //Métodos acessores
```

21 }  

---

## Classe Carro

O código abaixo mostra a implementação da classe `Carro`, que herda a estrutura e o comportamento da classe `Veiculo` (linha 1) e define suas características específicas, como o atributo `numPortas`. Além disso, esta classe sobreescreve o método `calculaImposto`, adicionando o valor fixo de R\$ 800,00. Neste caso, o método faz a chamada do cálculo do imposto da superclasse (através do `super`) e adiciona um comportamento adicional.

Observe que a mesma característica é observada no método construtor, onde o construtor da superclasse é chamado (linhas 10 e 14) com seus respectivos parâmetros, para então atribuir suas características específicas. Esta é uma boa prática, pois garante que modificações no construtor da superclasse não resultem em alterações nas subclasses.

```
1 public class Carro extends Veiculo {
2     private int numPortas;
3
4     @Override
5     public double calculaImposto() {
6         return super.calculaImposto() + 800;
7     }
8
9     public Carro() {
10        super();
11    }
12
13    public Carro(int numPortas, String marca, String modelo, String
14    ↪ cor, double valor) {
15        super(marca, modelo, cor, valor);
16        this.numPortas = numPortas;
17    }
18
19    //Métodos acessores
20 }
```

## Classe Moto

O código a seguir mostra a implementação da classe `Moto`, que também herda as características da classe `Veiculo` e adiciona um atributo específico (`cilindradas`). Neste caso, o cálculo do imposto para uma moto é igual ao cálculo geral. Por isso, não é necessária a sobreescrição do método `calculaImposto` nesta classe.

```
1 public class Moto extends Veiculo {
2     private int cilindradas;
```

```

3
4     public Moto() {
5         super();
6     }
7
8     public Moto(int cilindradas, String marca, String modelo, String
↪ cor, double valor) {
9         super(marca, modelo, cor, valor);
10        this.cilindradas = cilindradas;
11    }
12
13    //Métodos acessores
14 }

```

### 7.4.3. Uso das classes

Deseja-se implementar os seguintes métodos:

- Criação de registros de carros e motos e armazenamento em uma lista polimórfica.
- Verificação de veículos de uma determinada marca.
- Apresentação dos veículos e seus valores de impostos.
- Apresentação de todos os carros da lista.

Para isso, criaremos uma classe **Aplicacao**, responsável por armazenar a lista de veículos (objetos da classe **Veiculo** e, portanto, polimórfica) e implementar os métodos desejados. A Figura 7.5 mostra a estrutura de classes atualizada e o trecho de código da sequência apresenta a estrutura básica da classe aplicação.

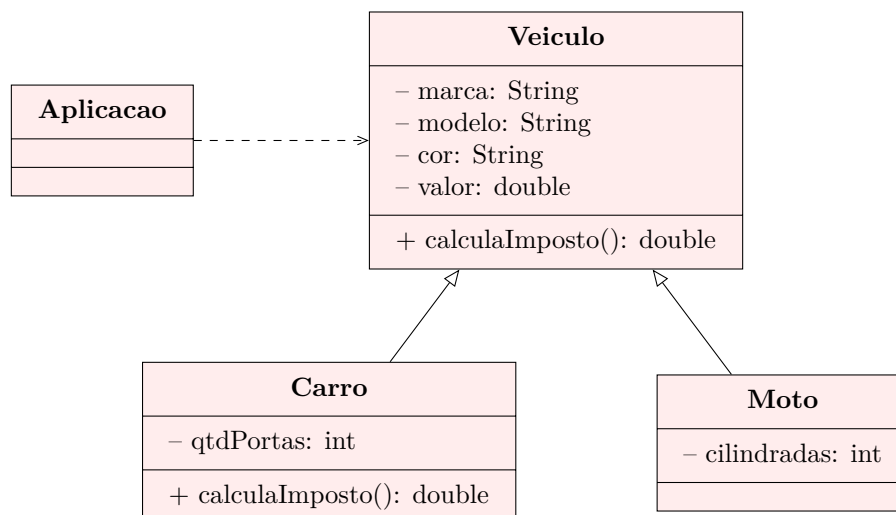


Figura 7.5.: Estrutura de classes atualizada para o exemplo dos veículos

```
1 public class Aplicacao {
2     private List<Veiculo> veiculos = new ArrayList<Veiculo>();
3
4     private void criaRegistros() {
5         // implementação omitida
6     }
7
8     private void veiculosDaMarca(String marca) {
9         // implementação omitida
10    }
11
12    private void mostraImpostos() {
13        // implementação omitida
14    }
15
16    private void mostraCarros() {
17        // implementação omitida
18    }
19 }
```

### Criação dos registros

O código abaixo mostra a implementação do método `criaRegistros`. São criados objetos do tipo `Carro` e `Moto` e adicionados na mesma lista (`veiculos`). Isso funciona pois a lista é de objetos da classe `Veiculo` (`List<Veiculo>`), que permite armazenar referências a qualquer classe que estenda `Veiculo`, que é o caso das classes `Carro` e `Moto`. Por conta disso, dizemos que a lista é polimórfica.

```
1 private void criaRegistros() {
2     Carro c1= new Carro(2, "VW", "Gol", "prata", 25000);
3     Carro c2 = new Carro(4, "Fiat", "Uno", "branco", 20000);
4     Carro c3= new Carro(4, "Renault", "Clio", "preto", 32000);
5     Carro c4= new Carro(2, "Fiat", "147", "amarelo", 8000);
6
7     Moto m1 = new Moto(150, "Honda", "CG", "azul", 7000);
8     Moto m2 = new Moto(150, "Yamaha", "YBR", "vermelho", 12000);
9
10    veiculos.add(c1);
11    veiculos.add(c2);
12    veiculos.add(c3);
13    veiculos.add(c4);
14    veiculos.add(m1);
15    veiculos.add(m2);
16 }
```



### Consulta de veículos de uma marca

O código abaixo mostra a implementação do método `veiculosDaMarca`. Nele, a lista de veículos é percorrida, independente da referência que se encontra a cada iteração (linha 3). Pela herança, é garantido que todos os objetos dessa lista possuam o método `getMarca`, que é utilizado para filtrar os registros e apresentar, ao final, a quantidade de veículos da marca recebida como parâmetro.

```
1 private void veiculosDaMarca(String marca) {
2     int qtd = 0;
3     for(Veiculo v: veiculos) {
4         if(v.getMarca().equals(marca))
5             qtd++;
6     }
7     JOptionPane.showMessageDialog(null, "A marca " + marca + " possui
↪ " + qtd + " veículos!");
8 }
```

#### SAÍDA:

```
A marca VW possui 1 veículos!
A marca Fiat possui 2 veículos!
A marca Renault possui 1 veículos!
A marca Honda possui 1 veículos!
A marca Yamaha possui 1 veículos!
```

### Apresentação dos impostos

O código abaixo mostra a implementação do método `mostraImpostos`. Em tempo de execução, o Java verifica qual a referência armazenada em `v` e executa o respectivo método `calculaImposto`. Ou seja, é executado o método implementado em `Veiculo` ou em `Carro`.

```
1 private void mostraImpostos() {
2     String texto = "";
3     for(Veiculo v : veiculos) {
4         texto += v.getMarca() + " " + v.getModelo() + "(" +
↪ v.getValor() + "): " + v.calculaImposto() + "\n";
5     }
6     JOptionPane.showMessageDialog(null, texto);
7 }
```

#### SAÍDA:

```
VW Gol (25000.0): 1300.0
Fiat Uno (20000.0): 1200.0
Renault Clio (32000.0): 1440.0
Fiat 147 (8000.0): 960.0
Honda CG (7000.0): 140.0
Yamaha YBR (12000.0): 240.0
```

## Apresentação dos carros

O código abaixo mostra a implementação do método `mostraCarros`. A lista é percorrida e a classe da referência armazenada em cada posição é verificada pelo método `instanceOf`. Este comando verifica se o objeto (à esquerda) é do tipo da classe desejada (à direita), retornando verdadeiro ou falso.

```
1 private void mostraCarros() {  
2     String texto = "";  
3     for(Veiculo v: veiculos) {  
4         if(v instanceof Carro)  
5             texto += v.getMarca() + " " + v.getModelo() + ", cor " +  
6             ↪ v.getCor() + "\n";  
7     }  
8     JOptionPane.showMessageDialog(null, texto);  
9 }
```

### SAÍDA:

```
VW Gol, cor prata  
Fiat Uno, cor branco  
Renault Clio, cor preto  
Fiat 147, cor amarelo
```

## 8. Classes abstratas

### 8.1. Conceitos e benefícios

O maior benefício oriundo da generalização/especialização de classes através da herança é o **polimorfismo**. Ele é um recurso poderoso, capaz de garantir ao sistema flexibilidade. Para estudar o conceito de classes abstratas e sua relação com o polimorfismo, consideremos um contexto de figuras geométricas, as quais compartilham entre si uma cor e possuem métodos para cálculo da sua área e do seu perímetro. A Figura 8.1 apresenta a estrutura de classes onde diferentes figuras geométricas herdam as características da classe **Figura**.

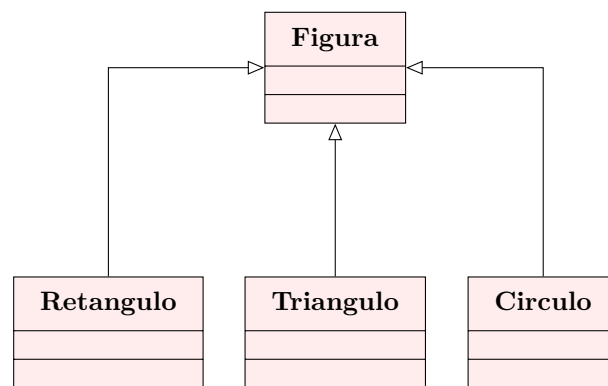


Figura 8.1.: Estrutura de classes para a modelagem de figuras geométricas

Os trechos de código a seguir mostram a implementação das classes **Figura** e **Retangulo**, envolvidas no diagrama apresentado pela Figura 8.1. A classe **Figura** define o atributo **cor** e os métodos **area** e **perimetro**, que serão herdados pelas subclasses. A classe **Retangulo** implementa alguns atributos adicionais, além de sobrescrever os métodos **area** e **perimetro**.

```
1 public class Figura {
2     private String cor;
3
4     public double area() {
5         return 0;
6     }
7
8     public double perimetro() {
9         return 0;
10    }
```

```
11 }

1 public class Retangulo extends Figura {
2     private double lado1, lado2;
3
4     public double area() {
5         return lado1 * lado2;
6     }
7
8     public double perimetro() {
9         return (lado1 * 2) + (lado2 * 2);
10    }
11 }
```

Neste exemplo, não faz sentido termos uma instância da classe `Figura`, pois uma figura é sempre um triângulo, um retângulo, um círculo, um trapézio, etc. A classe `Figura` define uma entidade abstrata, que só existe para definir uma estrutura comum a todas as figuras concretas, além de estabelecer os métodos que todas devem apresentar (cálculo da área e do trapézio).

Tecnicamente, usamos a classe `Figura` para garantir a estrutura e o comportamento de todas as figuras e para nos fornecer polimorfismo. Não faz sentido termos uma instância de `Figura` e calcularmos sua área ou perímetro, por exemplo.

Uma dica para identificar casos onde não faz sentido termos uma instância da superclasse é analisar seus métodos. Quando não é possível definir sua implementação, é um bom indicativo de que a classe não deve ser instanciada.

No exemplo das figuras geométricas, cada classe concreta (`Triangulo`, `Retangulo`, etc.) possui sua própria forma de calcular a área e o perímetro em função dos seus atributos. Isso não acontece na classe `Figura`, que é incapaz de prover uma implementação dos métodos supracitados.

Nestes casos, podemos definir a classe como **abstrata**, o que implica na impossibilidade de ser instanciada. Com isso, a classe se restringe a definir a estrutura e o comportamento das classes que a estenderem e fornecer polimorfismo.

Por exemplo, se tivermos uma lista de figuras (`List<Figura>`) estamos fazendo uso do polimorfismo. Caso a classe `Figura` seja abstrata, temos a garantia de que todos os objetos da lista são figuras concretas (triângulos, retângulos, etc.) e, com isso, faz sentido chamarmos os métodos `area` e `perimetro` para qualquer um deles.

## 8.2. Implementação

A palavra reservada `abstract` define uma classe como abstrata. O trecho de código abaixo mostra a definição da classe `Figura` como abstrata. Com isso, não é possível instanciar objetos dessa classe. O comando `Figura f = new Figura()` não com-

pilará, enquanto o comando `Figura f = new Circulo()` continua funcionando normalmente.

```
1 public abstract class Figura {  
2     private String cor;  
3  
4     public double area() {  
5         return 0;  
6     }  
7  
8     public double perimetro() {  
9         return 0;  
10    }  
11 }
```

No diagrama de classes, as classes abstratas devem ser representadas como tal, para que esta característica fique facilmente visível. Isso pode ser feito mantendo o nome da classe em *itálico* (ou ainda adicionando o estereótipo «**abstract**» à classe), conforme apresentado na Figura 8.2.

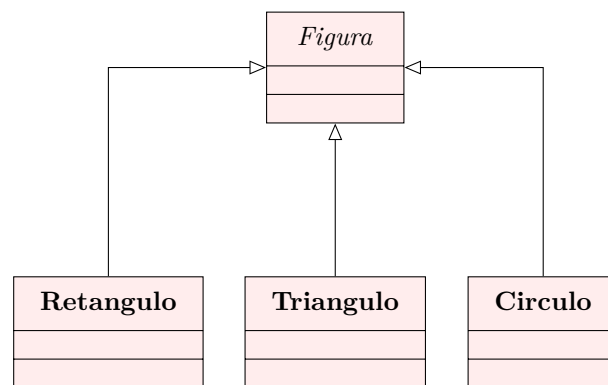


Figura 8.2.: Estrutura de classes para a modelagem de figuras geométricas com abstração

### 8.3. Métodos abstratos

Resolvemos o problema de não fazer sentido termos uma instância da classe `Figura`. Porém, a implementação padrão dos métodos `area` e `perimetro` ainda não fazem sentido na classe `Figura`, uma vez que trata-se de uma entidade abstrata. O ideal seria não haver implementação do método, uma vez que não faz sentido. Além disso, também seria ideal exigir que as subclasses implementem o método, uma vez que a implementação padrão não faz sentido para qualquer classe concreta.

Isso é possível definindo os métodos supracitados como abstratos. Com isso, apenas a assinatura do método é definida na superclasse, exigindo que as subclasses os implementem. Com isso, ao obtermos um objeto do tipo `Figura`, sabemos que ele é uma referência a um dos tipos concretos e que os métodos `area` e `perimetro` estão disponíveis (e fazem sentido).

Para definir um método abstrato, basta inserir a palavra **abstract** na sua assinatura e omitir sua implementação, colocando ponto-e-vírgula após o fechamento de parêntesis. O trecho de código abaixo mostra a implementação da classe **Figura** com a definição dos métodos abstratos. Logicamente, a classe abstrata poderia definir algum método com implementação padrão às suas subclasses, caso isso fizesse sentido (ocorre em outros contextos).

```
1 public abstract class Figura {  
2     private String cor;  
3     public abstract double area();  
4     public abstract double perimetro();  
5 }
```

É importante destacar que somente é possível definir um método como abstrato, caso ele pertence a uma classe abstrata. Classes concretas não podem conter métodos abstratos e devem implementar todos os métodos abstratos herdados. Uma classe abstrata pode definir métodos abstratos e métodos concretos simultaneamente.

## 9. Realização

### 9.1. Interfaces

Uma interface define um contrato ao qual uma classe pode assinar. Este contrato estabelece todos os métodos que esta classe deverá implementar e fornecer aos seus clientes. Quando uma classe assina o contrato (implementa a interface) deve implementar todos os métodos definidos nele.

Considerando o contexto de figuras geométricas apresentado pela Figura 9.1, a classe abstrata **Figura** define o atributo **cor** e os métodos **area** e **perimetro**. Se for necessário que cada figura implemente seu próprio método **desenhar**, uma boa estratégia consiste em definir o método abstrato **desenhar** na classe **Figura**, garantindo que cada subclasse forneça sua implementação, aproveitando-se do polimorfismo.

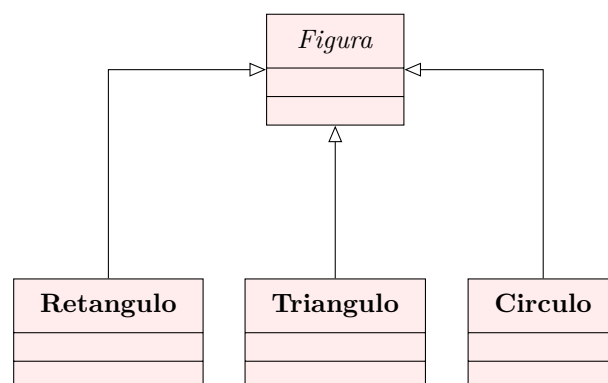


Figura 9.1.: Estrutura de classes para figuras desenháveis

Porém, o mesmo sistema possui as classes **Fonte**, **Serif** e **Grossa**, que definem as fontes de texto de um elemento gráfico. Estas classes também devem implementar seus métodos **desenhar**. Logo, o método de desenho não é exclusivo das figuras, portanto não fazem parte da sua classe.

A Figura 9.2 apresenta a situação do diagrama de classes incluindo a estrutura de classes das fontes. Diante disso, qual a forma mais adequada de estruturar o sistema para garantir que tanto as figuras quanto as fontes implementem o método **desenhar**? Este método deve estar definido em **Figura** ou **Fonte**? As figuras devem estender **Fonte**? As figuras e as fontes devem estender uma nova classe?

Adotar herança para resolver este problema não é a solução correta. Uma herança deve ser aplicada estritamente quando o relacionamento entre as classes responde a uma relação “**É UM**”. Neste caso, uma figura **não é uma** fonte, e uma fonte **não é uma** figura. Para resolver este problema, o ideal seria uma forma de apenas definir que as figuras e as fontes devem implementar o método **desenhar**. Isso é possível com uma **interface**!

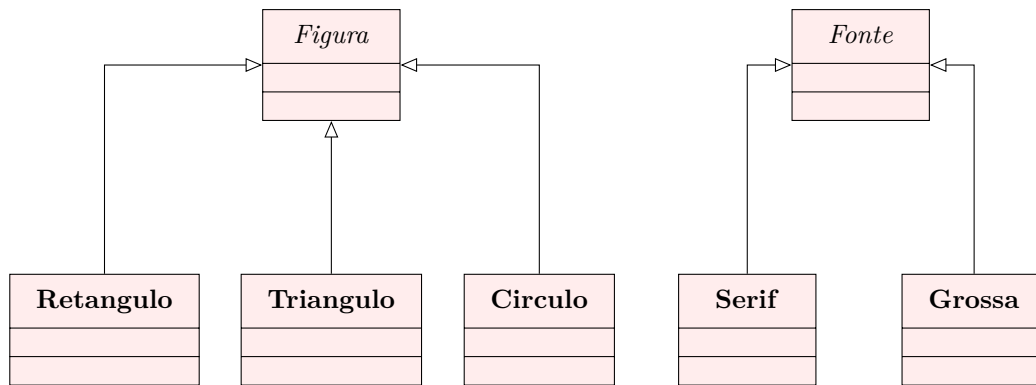


Figura 9.2.: Estrutura de classes para figuras e fontes desenháveis

Uma interface serve para isso, definir um contrato onde as classes que a realizam devem implementar os seus métodos. Com isso, podemos definir uma interface **Desenhavel** que exige que as classes que a implementem apresentem a implementação do método **desenha**. O trecho de código abaixo mostra a definição de uma interface.

```

1 public interface Desenhavel {
2     void desenha();
3 }

```

Basta então fazer com que as classes **Figura** e **Fonte** realizem (ou implementem) a interface criada, fazendo com que suas subclasses tenham que implementar o método de desenho. Logo, o uso de interfaces (ou realização) é diretamente aplicável quando queremos definir um comportamento aplicável a distintas classes, sem que estas tenham um vínculo semântico. Os códigos a seguir mostram estas definições.

```

1 public abstract class Figura implements Desenhavel {
2     private String cor;
3     public abstract double area();
4     public abstract double perimetro();
5     public void desenha() {
6         //implementação aqui.
7     }
8 }

```

```

1 public abstract class Fonte implements Desenhavel {
2     public void desenha() {
3         //implementação aqui.
4     }
5 }

```

Repare que, neste caso, as classes **Figura** e **Fonte** não são obrigadas a implementar o método **desenha**, pois são abstratas (o que não impede sua implementação). No entanto, neste caso a obrigatoriedade de implementação é passada às suas subclasses concretas.



Se a superclasse **Fonte** (por exemplo) implementar o método **desenhar**, as classes **Serif** e **Grossa** não precisam fazê-lo, a não ser que queiram sobrescrever o método. Se a classe **Fonte** não implementar o método, isso deve ser feito em **Serif** e **Grossa**.

A versão final do diagrama de classes é apresentada pela Figura 9.3. A interface é representada usando o estereótipo «interface». Perceba que entidades de diferentes naturezas (**Figura** e **Fonte**) implementam a interface. Ou seja, o contrato definido por ela serve para classes sem um vínculo semântico.

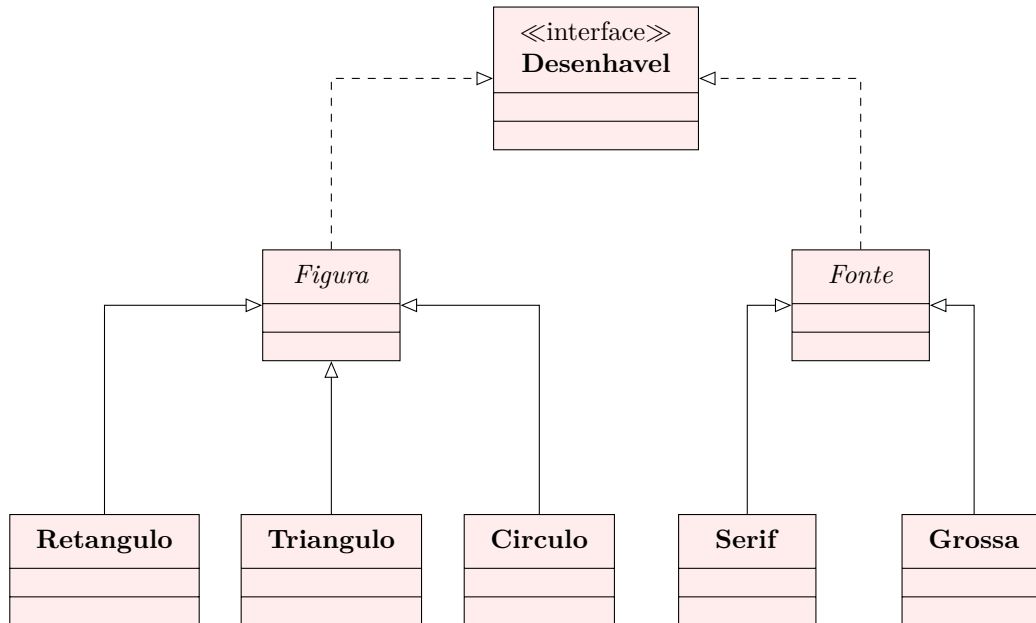


Figura 9.3.: Estrutura de classes para figuras e fontes com interface

**OBS:** a realização é representada na UML por uma linha tracejada e um triângulo não preenchido do lado da interface.

## 9.2. Mais polimorfismo

Com as interfaces, podemos usufruir de uma capacidade ainda maior de polimorfismo. Se tivermos uma lista de figuras (objetos de **Figura**), por exemplo, podemos nos referir a elas como desenháveis e chamar os métodos definidos na interface (contrato) **Desenhavel**. O trecho de código abaixo exemplifica este caso, onde ambos os laços de repetição são válidos.

```

1 public void run() {
2     List<Figura> figuras = criaListaFiguras();
3
4     for(Figura f: figuras) {
5         f.desenhar();
6     }
7
8     for(Desenhavel d: figuras) {

```

```
9      d.desenhar();  
10    }  
11 }
```

Se futuramente novas classes forem incluídas no sistema como subclasses de `Figura`, a aplicação continuará funcionando normalmente, uma vez que a herança e a realização garantem a existência do método `desenhar`.

Além disso, é possível criar métodos que operem sobre objetos que implementem a interface `Desenhavel`, independente da classe à qual o objeto pertença. Ou seja, as interfaces (realização) fornecem um nível ainda maior de flexibilidade. O trecho de código abaixo exemplifica o uso da realização para criação de rotinas polimórficas.

```
1  public void desenhaComponente(Desenhavel d) {  
2      d.desenha();  
3  }  
4  
5  public void desenhaComponentes(List<Desenhavel> lista) {  
6      for(Desenhavel d: lista) {  
7          d.desenha();  
8      }  
9  }
```

Se futuramente novas classes que implementem a interface `Desenhavel` forem incluídas no sistema, a aplicação continuará funcionando normalmente, uma vez que os métodos `desenhaComponente` e `desenhaComponentes` fazem uso do polimorfismo oriundo da realização.

## **Parte III.**

### **Tópicos adicionais**



## 10. Interfaces gráficas

A construção de interfaces gráficas se baseia no uso de componentes, que são objetos com capacidade de interagir com o usuário. O Java fornece bibliotecas de componentes para auxiliar na construção de interfaces gráficas. As bibliotecas mais conhecidas são o AWT (Abstract Window Toolkit) e o SWING. Este último é mais atual e possui mais componentes e recursos, mas ainda permite a utilização em conjunto com o anterior. Estas bibliotecas são disponibilizadas através dos pacotes `java.awt.*` e `javax.swing.*`.

Algumas IDEs possuem ferramentas para suporte à construção de interfaces gráficas, as quais permitem ao desenvolvedor construí-la de forma visual, arrastando seus componentes e implementando suas ações. Entre as IDEs mais conhecidas com suporte à construção de interfaces gráficas estão o NetBeans e o Eclipse. Este capítulo apresenta uma introdução ao desenvolvimento de aplicações gráficas usando o NetBeans. Desenvolveremos um sistema para cadastro de veículos (`CadastroVeiculo`).

### 10.1. Construção usando o NetBeans

O primeiro passo é criar um projeto do tipo `Aplicação Java` (`Java Application`). No pacote de códigos-fonte, clique com o botão direito e selecione as opções `Novo > Form JFrame`. Caso esta opção não seja apresentada, busque-a em `Outros`. Estes passos criarão uma classe que estende `JFrame`, que consiste em uma tela. Portanto, podemos chamar nossa primeira tela (classe) de `TelaPrincipal`. A Figura 10.1 ilustra o processo de criação da primeira tela do sistema.

O NetBeans fornece uma série de ferramentas para suporte à construção de interfaces gráficas. A Figura 10.2 mostra a IDE e suas ferramentas. No lado esquerdo, podemos acessar a estrutura do projeto e seus arquivos. Na parte central observamos a tela que está sendo construída. Nela podemos inserir componentes, posicioná-los e configurá-los. Na parte direita (acima) encontramos a paleta de componentes. Através dela é possível selecionar os componentes desejados e arrastá-los até a área de construção da interface. Logo abaixo observa-se as propriedades dos elementos. Basta selecionar um componente inserido na interface e suas propriedades são apresentadas, possibilitando a alteração e ajuste dos seus valores.

O componente `JFrame`, que consiste em uma tela através do qual o usuário interage com o sistema, é criado automaticamente com um método `main`, permitindo que a aplicação inicie por ele. O ideal é que uma classe de controle seja responsável por iniciar o sistema e criar as suas telas. Porém, por questões de simplicidade, utilizaremos a `TelaPrincipal` como ponto de partida da aplicação. Logo, a classe principal criada no projeto pode ser excluída. A Figura 10.3 mostra a estrutura atual do projeto.

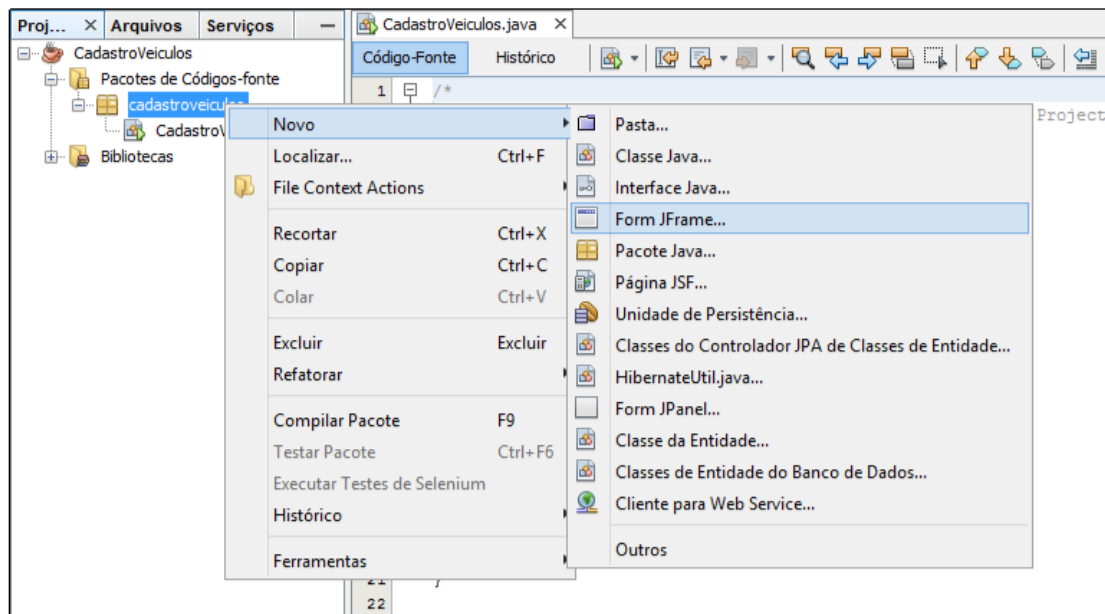


Figura 10.1.: Criação de uma tela (JFrame)

Logo acima da pré-visualização da interface encontram-se algumas abas (chamadas visões), entre elas **Código-fonte** e **Projeto**. Ao selecionar a aba **Projeto**, as ferramentas de construção visuais são apresentadas. Ao selecionar a aba **Código-fonte**, o código gerado é apresentado e o programador pode fazer as alterações desejadas e a implementação dos métodos da aplicação.

Abaixo é apresentado o código para a tela inicial criada. Perceba que a classe estende **JFrame**. O método **main** cria a janela e a torna visível. No construtor, todos os componentes inseridos na tela são inicializados. Nesta classe podemos definir nossas variáveis, como a lista de veículos e os objetos necessários à sua manipulação.

```

1  public class TelaPrincipal extends javax.swing.JFrame {
2
3      public TelaPrincipal() {
4          initComponents();
5      }
6
7      @SuppressWarnings("unchecked")
8      //Generated code
9
10     public static void main(String args[]) {
11         java.awt.EventQueue.invokeLater(new Runnable() {
12             public void run() {
13                 new TelaPrincipal().setVisible(true);
14             }
15         });
16     }
17 }

```

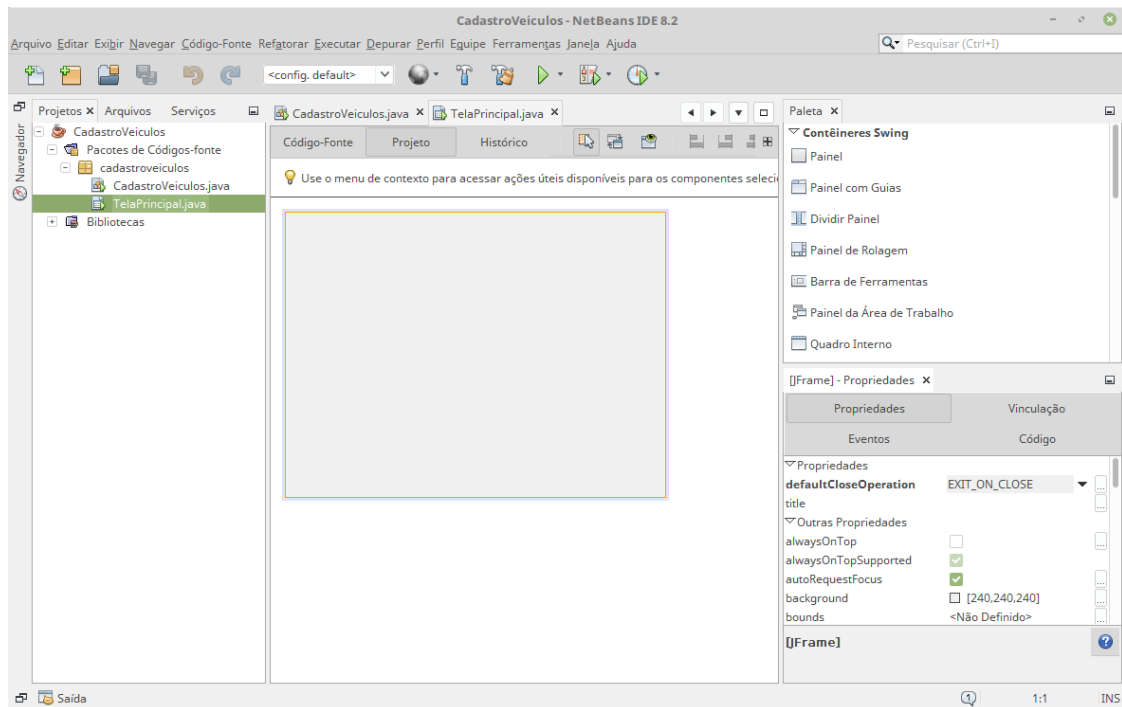


Figura 10.2.: NetBeans com ferramentas para interfaces gráficas

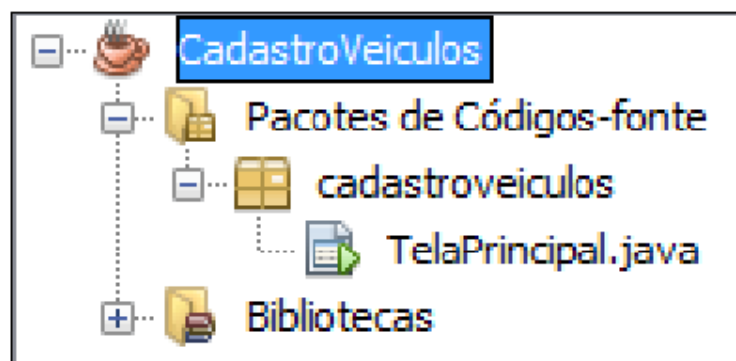


Figura 10.3.: Estrutura inicial do projeto

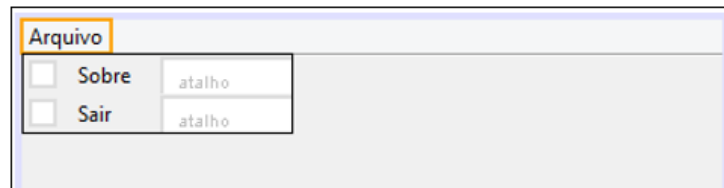
## 10.2. Barra de menu, eventos e ações

Utilizando a visão de **Projeto**, na categoria **Menus Swing** da paleta de componentes, arraste o elemento **Barra de Menu** para a janela. Será criado um menu com as opções **File** e **Edit**. Clicando com o botão direito sobre estes itens é possível:

- Editar o texto que é apresentado em tela.
- Alterar o nome da variável (objeto) do componente.
- Adicionar itens a esta opção do menu (**Adicionar da Paleta > Item de Menu**).
- Para cada item adicionado, é possível realizar as mesmas ações.

Com o uso destas opções, crie configuração do menu apresentada na Figura 10.4.

**OBS:** é importante definirmos nomes sugestivos a cada objeto que será manipulado,

Figura 10.4.: Menu para a aplicação `CadastroVeiculos`

de forma a dar legibilidade ao código. Exemplos: `menuArquivo`, `menuSobre` e `menuSair`.

Execute a aplicação e veja o resultado, que deve ser parecido com o apresentado na Figura 10.5. O título da tela (“Cadastro de Veículos”) pode ser definido na propriedade `title` da janela. Selecionando a janela, suas propriedades são apresentadas na parte direita da interface. Basta, portanto, localizar e alterar a propriedade desejada.

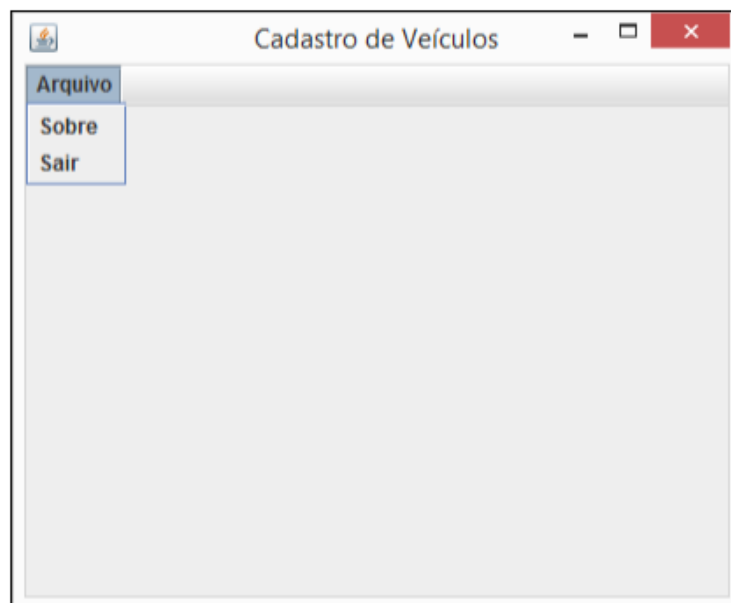


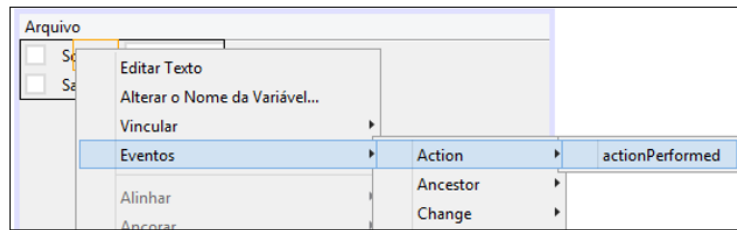
Figura 10.5.: Aplicação inicial com barra de menu

do o usuário clicar na opção **Sobre**, devemos apresentar uma janela com as informações de desenvolvimento da aplicação. Podemos apresentar uma caixa de diálogo (`JOptionPane`). Logo, basta definirmos a exibição da janela no evento de ação do item de menu correspondente. Ou seja, quando o usuário clica no item, o método (chamado evento) é disparado.

Para isso, clique com o botão direito no item **Sobre** e selecione as opções **Eventos** > **Action** > `actionPerformed`. A Figura 10.6 mostra estas ações. Com isso, o NetBeans criará um método que é invocado sempre que o item de menu for selecionado pelo usuário. Nele, podemos implementar o comportamento desejado.

O trecho de código abaixo mostra a implementação do método que é disparado no evento de ação do componente `menuSobre`. Quando o usuário clicar neste item, será apresentado uma caixa de diálogo com informações do desenvolvimento. Repare que, uma vez que temos uma janela no sistema, podemos defini-la como componente pai da caixa de diálogo.



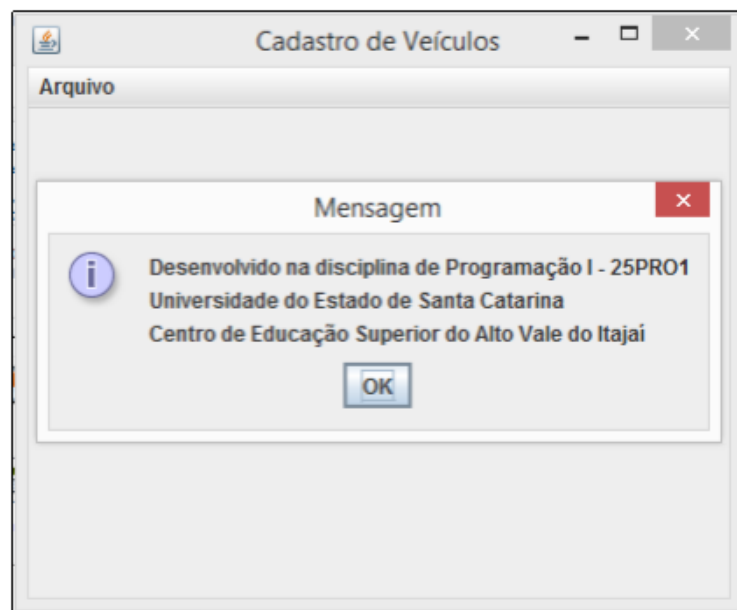
Figura 10.6.: Definição do evento `actionPerformed`

Isso é feito no primeiro argumento do método `showMessageDialog`. A Figura 10.7 mostra o resultado da execução do evento de clique no item de menu **Sobre**.

```

1 private void menuSobreActionPerformed(java.awt.event.ActionEvent evt)
2     {
3         JOptionPane.showMessageDialog(this,
4             "Desenvolvido na disciplina de Programação I - 25PRO1\n"
5             + "Universidade do Estado de Santa Catarina\n"
6             + "Centro de Educação Superior do Alto Vale do Itajaí");
7     }

```

Figura 10.7.: Execução da ação do menu **Sobre**

Faremos o mesmo procedimento para definir o código a ser executado quando o usuário clicar no item **Sair**. Neste caso, a tarefa é fechar a janela e finalizar a aplicação, mostrando uma mensagem de encerramento. A finalização da aplicação é obtida pelo método `dispose`. O trecho de código abaixo mostra sua implementação, enquanto a Figura ?? mostra o resultado da sua execução.

```

1 private void menuSairActionPerformed(java.awt.event.ActionEvent evt)
  ↪ {
2     JOptionPane.showMessageDialog(this, "Encerrando a aplicação...");
3     dispose();
4 }

```

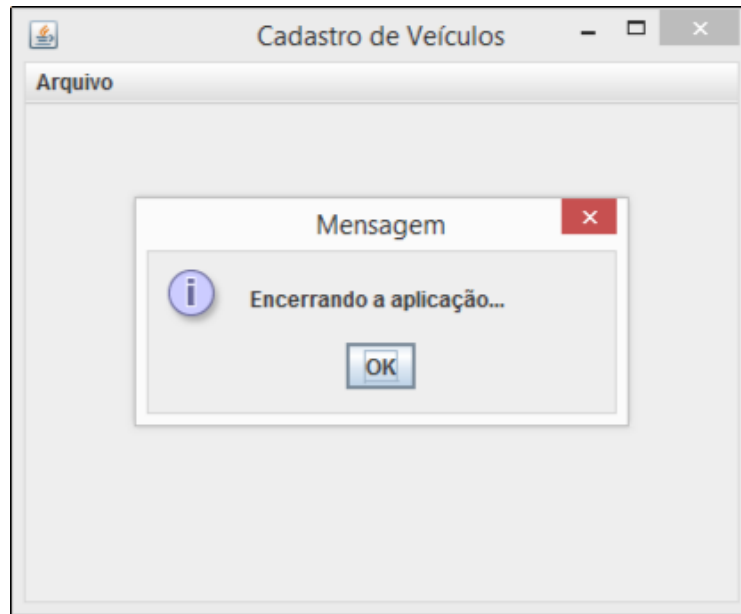


Figura 10.8.: Execução da ação do menu Sair

### 10.3. Entidade Veiculo

Como a aplicação tem como objetivo o cadastro de veículos, precisamos definir a entidade **Veiculo**. Esta classe possui modelo, marca e ano, bem como seus métodos construtores e acessores. A Figura 10.9 mostra sua representação UML e o código correspondente é apresentado na sequência.

Veiculo
– modelo: String – marca: String – ano: int
+ métodos construtores + métodos acessores

Figura 10.9.: Representação UML da classe **Veiculo**

```

1 public class Veiculo {
2     private String modelo;
3     private String marca;

```

```

4     private int ano;
5
6     //Métodos construtores e acessores
7 }

```

## 10.4. Componentes da tela

A tela principal deve apresentar um formulário através do qual o usuário poderá cadastrar, consultar, alterar e excluir veículos. Os componentes disponíveis na paleta (alguns deles) são apresentados pela Figura 10.10. Para construir a interface desejada, utilizaremos os componentes listados abaixo (entre parêntesis é apresentada a classe que implementa cada componente).

- Label (JLabel): rótulo ou saída de texto na tela.
- TextField (JTextField): campo de entrada de texto.
- Button (JButton): botão de ação.



Figura 10.10.: Paleta de componentes Swing

A Figura 10.11 apresenta o formulário formado pelos componentes supracitados. Do lado esquerdo podemos observar os elementos Label, que formam os textos *Modelo.:*, *Marca.:* e *Ano.:*. Os campos de entrada de texto são componentes **TextField** e os botões são componentes **Button**. Assim como nos itens de menu, estes objetos devem receber um nome sugestivo e seu texto de apresentação deve ser definido nas propriedades. Posicione cada elemento e defina os textos de exibição e nomes sugestivos aos objetos, seguindo a proposta da Figura 10.11.

## 10.5. Funcionalidades

Para implementar as funcionalidades, a classe **TelaPrincipal** deverá manter uma lista de veículos (onde os registros serão armazenados) e um objeto da classe **Veiculo** (que

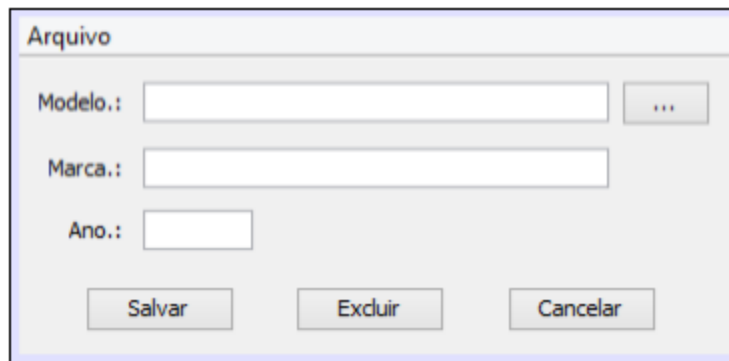
A imagem mostra uma janela de diálogo intitulada "Arquivo". Ela contém três campos de texto rotulados "Modelo.:", "Marca.:" e "Ano.:". O campo "Modelo.:" possui um botão de menu (três pontos) à sua direita. Na base da janela, há três botões: "Salvar", "Excluir" e "Cancelar".

Figura 10.11.: Formulário da tela principal

representará o registro manipulado pelo usuário).

```
1 public class TelaPrincipal extends javax.swing.JFrame {  
2  
3     private List<Veiculo> listaVeiculos = new ArrayList<Veiculo>();  
4     private Veiculo veiculo = new Veiculo();  
5  
6     //...  
7 }
```

### 10.5.1. Salvar registro

Da mesma forma como é feito com os menus, é possível definir métodos que serão executados quando os botões forem clicados. O processo é o mesmo, definindo um evento do tipo `actionPerformed`.

O botão **Salvar** serve para gravar um novo registro na lista, com base nas informações digitadas pelo usuário, ou gravar as alterações feitas pelo usuário em um objeto previamente recuperado da lista. Logo, os passos que devem ser implementados são:

- Recuperar os valores digitados pelo usuário no formulário.
- Atualizar os atributos do objeto `veiculo` com os valores recuperados.
- Se o objeto não se encontra na lista, é um novo registro e deve ser incluído na mesma. Caso contrário, trata-se de uma alteração e a atualização do objeto já é suficiente.
- Instanciar um novo objeto em `veiculo`, permitindo o cadastro de um novo registro.
- Limpar os campos do formulário para que o usuário possa cadastrar um novo registro.

O trecho de código abaixo mostra a implementação resultante do método de salvar. Repare que isto é feito no evento de ação do componente `btSalvar`. As linhas 2 a 4 recuperam os valores informados pelo usuário na tela. Perceba que é possível acessar os objetos dos componentes, recuperando o valor digitado através do método `getText`. As

linhas 6 a 8 setam os valores recuperados aos atributos do objeto `veiculo`. Nas linhas 10 a 12, se o objeto não se encontra na lista, trata-se de um novo registro e ele deve ser inserido na mesma. Caso contrário, trata-se de uma alteração de um registro existente (pois já encontra-se na lista) e a atualização do objeto já é suficiente. A linha 13 instancia um novo objeto para permitir o cadastro de um novo registro. Finalmente, as linhas 15 a 17 limpam os campos da tela após o cadastro ou alteração.

```
1 private void btSalvarActionPerformed(java.awt.event.ActionEvent evt)
   ↳ {
2     String modelo = edModelo.getText();
3     String marca = edMarca.getText();
4     int ano = Integer.parseInt(edAno.getText());
5
6     veiculo.setModelo(modelo);
7     veiculo.setMarca(marca);
8     veiculo.setAno(ano);
9
10    if(!listaVeiculos.contains(veiculo)) {
11        listaVeiculos.add(veiculo);
12    }
13    veiculo = new Veiculo();
14
15    edModelo.setText("");
16    edMarca.setText("");
17    edAno.setText("");
18 }
```

### 10.5.2. Cancelar preenchimento

O botão Cancelar deve limpar todos os campos. Caso os valores nos campos sejam referentes a um objeto pesquisado pelo usuário, a referência armazenada em `veiculo` deve ser removida e uma nova instância atribuída ao objeto. Isso é feito no método apresentado abaixo, vinculado ao botão `btCancelar`.

```
1 private void btCancelarActionPerformed(java.awt.event.ActionEvent
   ↳ evt) {
2     edModelo.setText("");
3     edMarca.setText("");
4     edAno.setText("");
5     veiculo = new Veiculo();
6 }
```

### 10.5.3. Excluir registro

O botão Excluir remove da lista de veículos um registro pesquisado anteriormente. Como o registro pesquisado se encontra armazenado no objeto `veiculo`, basta removê-lo da lista, atribuir uma nova instância a ele e limpar os campos da tela. Isso é feito no método apresentado abaixo, vinculado ao botão `btExcluir`.

```
1 private void btExcluirActionPerformed(java.awt.event.ActionEvent evt)
   ↳ {
2     listaVeiculos.remove(veiculo);
3     this.veiculo = new Veiculo();
4
5     edModelo.setText("");
6     edMarca.setText("");
7     edAno.setText("");
8 }
```

### 10.5.4. Pesquisar registro

O botão Pesquisar permite recuperar um objeto da lista de veículos e preencher os campos do formulário com os valores dos seus atributos. Essa pesquisa é feita com base no valor de modelo informado pelo usuário, por isso o botão se localiza ao lado deste campo.

Caso o veículo seja encontrado, sua referência deve ser armazenada no objeto `veiculo` e os campos do formulário são preenchidos com os valores dos seus atributos. Com isso, caso o usuário modifique os valores dos campos e clique em **Salvar**, o objeto da lista de veículos será atualizado, ao invés de incluído um novo objeto. Caso o usuário clique em **Excluir**, o objeto será removido da lista. Caso o usuário clique em **Cancelar**, nada é realizado e os campos e o objeto são “reiniciados”.

O código vinculado ao evento de ação do botão **Pesquisar** é apresentado abaixo. Na linha 2, o modelo digitado pelo usuário é recuperado. A lista de veículos é percorrida e, caso encontrado um objeto com o modelo buscado, sua referência é armazenada em `veiculo` e seus atributos são inseridos nos campos do formulário (linhas 4 a 11).

```
1 private void btPesquisarActionPerformed(java.awt.event.ActionEvent
   ↳ evt) {
2     String modelo = edModelo.getText();
3
4     for(Veiculo v : listaVeiculos) {
5         if(v.getModelo().equals(modelo)) {
6             veiculo = v;
7             edModelo.setText(veiculo.getModelo());
8             edMarca.setText(veiculo.getMarca());
9             edAno.setText(String.valueOf(veiculo.getAno()));
```

```
10         break;
11     }
12 }
13 }
```

### 10.5.5. Excluir registro

Uma característica importante do botão **Excluir** é o fato de ele não fazer sentido quando o usuário está cadastrando um novo veículo. Sua funcionalidade só pode ser executada quando o usuário carregar um veículo previamente cadastrado. Para evitar problemas, podemos desabilitar o botão quando o mesmo não deve ser clicado, impedindo que o usuário execute o método vinculado a ele e deixando claro sua impossibilidade de uso.

Isso pode ser feito modificando a propriedade `enabled` através do método `setEnabled(true)` ou `setEnabled(false)`, habilitando e desabilitando o componente. O trecho de código abaixo apresenta um exemplo do uso deste método.

```
1 btExcluir.setEnabled(true);
2 btExcluir.setEnabled(false);
```

O botão **Excluir** deve ser habilitado nos seguintes casos:

- Quando a ação de pesquisa encontra o objeto e o carrega na tela, permitindo ao usuário excluir o veículo, se desejado.

O botão **Excluir** deve ser desabilitado nos seguintes casos:

- Quando a ação de salvar é realizada, pois pode se tratar de um objeto carregado da lista, com suas alterações sendo salvas.
- Quando a ação de cancelar é realizada, pois pode se tratar de um objeto carregado da lista, onde o usuário está cancelando sua alteração.
- Quando a ação de exclusão é realizada, pois ao concluir a exclusão de um registro, uma nova instância é definida e não é possível excluir novamente.
- Quando a aplicação inicia, pois o botão deve estar desabilitado por padrão. Isso pode ser feito no construtor da classe, após a inicialização dos seus componentes (trecho de código abaixo).

```
1 public TelaPrincipal() {
2     initComponents();
3     btExcluir.setEnabled(false);
4 }
```

## 10.6. Outros componentes

Existem muitos outros componentes swing para a construção de interfaces gráficas interativas. A Figura 10.12 apresenta a aplicação de componentes de *checkbox*, *radio button*, menu de seleção, campo de senha, área de texto e caixa de seleção.

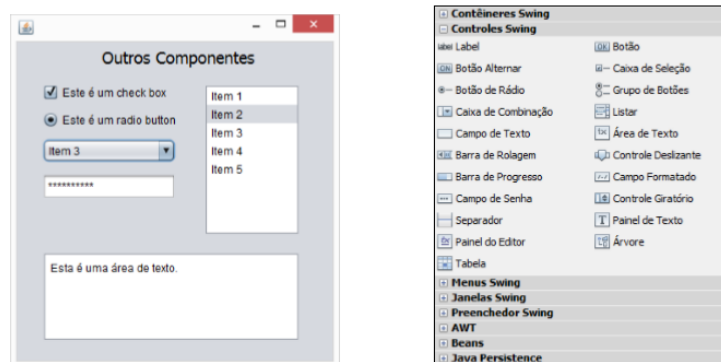


Figura 10.12.: Outros componentes swing

Para consultar a documentação dos componentes que deseja utilizar, consulte <http://docs.oracle.com/javase/8/docs/api>.



# 11. Persistência em arquivos

Aplicações precisam de algum mecanismo de persistência de dados, pois informações armazenadas na memória são perdidas sempre que a aplicação é encerrada. A forma mais simples de fazer isso em Java é salvar os dados em arquivos. As operações de leitura e escrita de dados em arquivos são feitas através das bibliotecas de entrada e saída do Java (ou bibliotecas I/O – Input/Output). A Figura 11.1 mostra o esquema geral da biblioteca de entrada e saída que será usada neste capítulo. O programa em Java comunica-se com as classes `FileReader` e `FileWriter` para as operações de leitura e gravação de dados no arquivo.

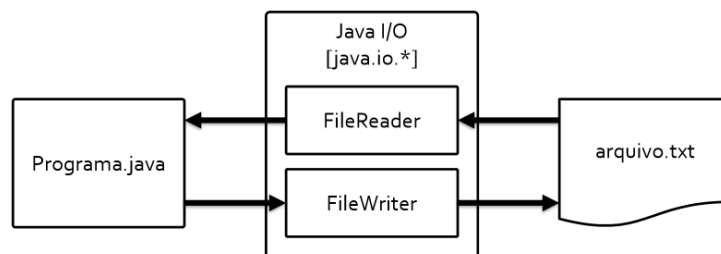


Figura 11.1.: Esquema de classes para persistência em arquivos

## 11.1. Operação de escrita

A classe `FileWriter` fornece os métodos necessários para a escrita de dados em arquivos. Na sua criação, é passado o nome do arquivo e o argumento `append`. Se verdadeiro, caso existam informações no arquivo, elas são mantidas e os novos dados são inseridos no final do arquivo. Se falso, caso existam informações no arquivo, elas são apagadas e os novos dados são inseridos no seu lugar.

O trecho de código abaixo mostra o uso da classe `FileWriter` para escrita de dados. O método `write` recebe como parâmetro o valor e o escreve no arquivo vinculado ao `writer` (neste exemplos, `peessoas.txt`). Repare que é necessário fechar o `writer` após o uso, bem como circundar o código com instruções `try-catch` para tratar o disparo de exceções no acesso ao arquivo em disco.

```
1 try {
2     FileWriter writer = new FileWriter("peessoas.txt", true);
3     writer.write("Este texto será inserido no arquivo");
4     writer.write("\n");
5     writer.close();
}
```

```
6 } catch (IOException ex) {  
7     Logger.getLogger(  
8         PersistenciaArquivos.class.getName()).log(Level.SEVERE, null,  
9         ex);  
10 }
```

## 11.2. Operação de leitura

A classe `FileReader` fornece métodos para abertura de arquivos e leitura dos seus dados. Porém, seus métodos permitem apenas a leitura de caracteres. Para ler os dados por linha, utilizaremos um objeto da classe `BufferedReader`. Na criação, instanciamos um novo `FileReader` passando o nome do arquivo.

O trecho de código mostra a implementação do método de leitura de dados usando as classes supracitadas. O método `readLine` faz a leitura de toda a linha e passa para a próxima. Após ler a última linha do arquivo, o método devolve `null`. Por isso, enquanto a linha lida for diferente de nulo, a linha é impressa em tela. Após o término, o `reader` é fechado. Assim como os métodos de escrita, o código deve ser circundado com as instruções `try-catch`.

```
1 try {  
2     BufferedReader reader = new BufferedReader(new  
3         ↪ FileReader("pessoas.txt"));  
4     String linha;  
5     while((linha = reader.readLine()) != null) {  
6         System.out.println(linha);  
7     }  
8     reader.close();  
9 } catch (FileNotFoundException ex) {} catch (IOException ex) {}
```

## 11.3. Exemplo – Cadastro de Pessoas

Considere um sistema para cadastro de pessoas com persistência dos dados em arquivos texto. O sistema deve fornecer as operações de cadastro, exclusão, consulta de uma pessoa pelo seu nome e consulta de todas as pessoas cadastradas.

### 11.3.1. Entidade Pessoa

O trecho de código abaixo mostra a classe `Pessoa`, que define a entidade que será cadastrada e persistida. Repare no método `toWriteString` (linhas 8 a 10), que retorna o texto que será usado para armazenar a pessoa no arquivo, com cada atributo separado pelo caracter `;`. O método `toString`, por sua vez, retorna o texto que deve ser usado para apresentar o objeto em tela.

```
1 public class Pessoa {
2     private String nome;
3     private int idade;
4     private char sexo;
5
6     //Métodos construtores omitidos
7
8     public String toWriteString() {
9         return nome + ";" + idade + ";" + sexo;
10    }
11
12    public String toString() {
13        String genero;
14        if(sexo == 'M') genero = "masculino";
15        else genero = "feminino";
16        return nome + ", possui " + idade + " anos de idade e é do sexo
17    ↪ " + genero + ".";
18    }
19
20    //Métodos acessores omitidos
21 }
```

### 11.3.2. Classe para persistência de dados

O trecho de código abaixo mostra a classe `PersistenciaArquivos`, cuja responsabilidade é fornecer métodos para as operações desejadas (cadastro, exclusão e consulta de pessoas). O método `inserir` armazena uma pessoa no arquivo. O método `ler` recupera uma pessoa do arquivo pelo seu nome, recebido como argumento. O método `lerTodasPessoas` devolve uma lista com todas as pessoas armazenadas no arquivo. O método `excluir` remove uma pessoa do arquivo pelo seu nome, recebido como argumento.

```
1 public class PersistenciaArquivos {
2     public static void excluir(String nome) {
3         //...
4     }
5
6     public static List<Pessoa> lerTodasPessoas() {
7         //...
8     }
9
10    public static Pessoa ler(String nome) {
11        //...
12    }
13
14    public static void inserir(Pessoa p){
```

```
15     //...
16 }
17 }
```

### 11.3.3. Escrita de uma pessoa

O método `inserir` recebe um objeto da classe `Pessoa` e o armazena no arquivo. Seu código é apresentado abaixo. Neste processo, o método armazena no arquivo o conteúdo retornado pelo método `toStringFile`. Cada pessoa é escrita em uma linha, portanto ao final de cada pessoa é armazenada uma quebra de linha (`\n`).

```
1 public static void inserir(Pessoa p){
2     try {
3         FileWriter writer = new FileWriter("pessoas.txt", true);
4         writer.write(p.toStringFile());
5         writer.write("\n");
6         writer.close();
7     } catch (IOException ex) {
8         Logger.getLogger(PersistenciaArquivos.class.getName()).log(
9             Level.SEVERE, null, ex);
10    }
11 }
```

### 11.3.4. Leitura de uma pessoa

O código abaixo mostra a implementação do método `ler`, que recebe o nome de uma pessoa (`String`) e retorna um objeto da classe `Pessoa` com a referência buscada, caso encontrado. Caso contrário, o método retorna `null`. O arquivo é lido linha por linha, ou seja, enquanto o retorno do método `readLine` for diferente de `null` (linha 5). Cada linha é dividida pelo caracter separador (`;`), de modo a obter o valor de cada atributo da pessoa armazenada no arquivo. Esta operação é feita pelo método `split` (linha 6 – detalhado abaixo). Caso o nome da pessoa seja igual ao nome buscado, a pessoa é recuperada (linha 7). Nas linhas 8 a 13 o objeto `Pessoa` é criado e seus atributos são recuperados, para então retorná-lo.

```
1 public static Pessoa ler(String nome) {
2     try {
3         BufferedReader reader = new BufferedReader(new
4         ↪ FileReader("pessoas.txt"));
5         String linha;
6         while((linha = reader.readLine()) != null) {
7             String[] conteudo = linha.split(";");
8             if(conteudo[0].equals(nome)) {
9                 Pessoa p = new Pessoa();
10            }
11        }
12    }
13 }
```

```

9         p.setNome(conteudo[0]); //nome na [0]
10        p.setIdade(Integer.parseInt(conteudo[1])); //idade na [1]
11        p.setSexo(conteudo[2].charAt(0)); //sexo [2]
12        reader.close();
13        return p;
14    }
15 }
16 reader.close();
17 } catch (FileNotFoundException ex) {} catch (IOException ex) {}
18 return null;
19 }

```

O método `split` divide a `String` conforme o caracter recebido (no exemplo, é usado o caracter `';``'`). Ele separa cada valor e adiciona em uma posição de um vetor de `String`, retornando-o. O código abaixo exemplifica o uso do método `split`.

```

1 String texto = "Este texto;está separado;para
   ↳ posterior;recuperação!";
2 String[] conteudo = texto.split(";");

```

**Resultado:**

```

conteudo[0] "Este texto"
conteudo[1] "está separado"
conteudo[2] "para posterior"
conteudo[3] "recuperação!"

```

### 11.3.5. Leitura de todas as pessoas

O trecho de código a seguir mostra a implementação do método `lerTodasPessoas`, que faz a leitura de todos os registros de pessoas armazenados no arquivo, retornando uma lista de objetos da classe `Pessoa`. A leitura é igual à de uma pessoa, mas todas as linhas são lidas, atribuídas ao objeto `Pessoa` e incluídos na lista, que é devolvida ao final do método.

```

1 public static List<Pessoa> lerTodasPessoas() {
2     List<Pessoa> listaPessoas = new ArrayList<Pessoa>();
3     try {
4         BufferedReader reader = new BufferedReader(new
   ↳ FileReader("pessoas.txt"));
5         String linha;
6         while((linha = reader.readLine()) != null) {
7             String[] conteudo = linha.split(";");
8             Pessoa p = new Pessoa();
9             p.setNome(conteudo[0]);
10            p.setIdade(Integer.parseInt(conteudo[1]));
11            p.setSexo(conteudo[2].charAt(0));

```

```
12     listaPessoas.add(p);
13 }
14 reader.close();
15 } catch (FileNotFoundException ex) {} catch (IOException ex) {}
16 return listaPessoas;
17 }
```

### 11.3.6. Exclusão de uma pessoa

Como não existe um método para excluir uma linha única de um arquivo, a estratégia de exclusão consiste em ler todos os dados, apagar todo o arquivo e reescrever todos os registros novamente, exceto aquele que deve ser excluído. O trecho de código abaixo mostra a implementação do método `excluir`. Repare que o método carrega uma lista com todas as pessoas (linha 2) e a percorre. Apenas os dados da pessoa buscada não são inseridos na `String` a ser regravada (linhas 6 e 7). Após lidas todas as pessoas, o arquivo é aberto com modo `append = false`, que apaga todos os registros previamente armazenados, para então gravar os novos dados.

```
1  public static void excluir(String nome) {
2      List<Pessoa> todasPessoas = lerTodasPessoas();
3      String conteudo = "";
4
5      for(Pessoa p : todasPessoas) {
6          if(!p.getNome().equals(nome))
7              conteudo += p.toString() + "\n";
8      }
9
10     try {
11         FileWriter writer = new FileWriter("pessoas.txt", false);
12         writer.write(conteudo);
13         writer.close();
14     } catch (IOException ex) {
15         Logger.getLogger(PersistenciaArquivos.class.getName()).log(
16             Level.SEVERE, null, ex);
17     }
18 }
```

**OBS:** uma estratégia similar pode ser adotada na edição de dados, onde todos os registros são reescritos, atualizando os atributos do registro buscado.

## **Parte IV.**

### **Apêndices**





# A. Lista de exercícios

## 1. Revisão sobre Java

### Exercício 1.1. (Salario)

Uma empresa decide dar um aumento de 25,5% aos funcionários cujo salário é inferior a R\$ 2.000,00 e tenha mais de 2 dependentes, 15% para os que ganham acima ou igual de R\$ 2.000,00 e tenham um dependente e 7,5% para os que ganham acima de R\$ 3.000,00 e não tenham dependente. Escreva um algoritmo que leia as informações de um funcionário e informe seu salário reajustado conforme regras.

### Exercício 1.2. (Bhaskara)

Escreva um programa para resolver equações do segundo grau. O algoritmo deverá ler os coeficientes A, B e C e resolver a equação conforme instruções abaixo.

$$\frac{x = -b \pm \sqrt{b^2 - 4ac}}{2a}$$

Identifique os casos onde não existe uma raiz real (interior da raiz quadrada negativo), existe uma única raiz real (interior da raiz quadrada igual a zero) e quando existem duas raízes reais (interior da raiz quadrada maior que zero). Apresente em tela a(s) raiz(es) da função.

### Exercício 1.3. (NumeroPrimo)

Escreva um programa que leia um numero inteiro positivo e diga se ele é um número primo. O programa deverá ser executado para quantos números o usuário desejar. O usuário pode finalizar a execução do algoritmo informando -1.

### Exercício 1.4. (MediaTurma)

Na disciplina de programação, são realizadas três provas ao longo do semestre. Escreva um algoritmo que leia as três notas de cada um dos alunos (o número de alunos na disciplina é informado no início da execução do algoritmo). Considerando a média aritmética simples das três notas, o algoritmo deve informar a quantidade de alunos aprovados, de alunos em exame e de alunos reprovados. Para aprovar a disciplina a média deve ser maior ou igual a 7. Uma média maior ou igual a 3 e menor que 7 deixa o aluno em exame, enquanto uma média menor que 3 implica na reprovação do aluno.

### Exercício 1.5. (Calculos)

Crie um programa que leia um valor inteiro positivo. Crie três métodos para realização de cálculos sobre o valor informado. O primeiro método recebe o valor como parâmetro e retorna seu fatorial. O segundo método recebe o valor como parâmetro e calcula a soma de todos os números naturais menores ou iguais ao valor informado. O terceiro

Salário	Dependentes	Benefício
Até R\$ 1.500,00	Até 2	R\$ 500,00
	Mais que 2	R\$ 800,00
Acima de R\$ 1.500,00	Até 2	R\$ 300,00
	Mais que 2	R\$ 500,00

método recebe o valor como parâmetro e imprime todos os números ímpares entre o valor informado e 50.

### Exercício 1.6. (BeneficioFuncionarios)

Uma empresa pretende ofertar um benefício aos funcionários carentes. Funcionários com pelo menos 10 anos de serviço e pelo menos 30 anos de idade têm o direito de receber o benefício. O valor pago a eles é calculado em função do salário e da quantidade de dependentes (tabela abaixo). Escreva um algoritmo que leia o tempo de serviço do funcionário, sua idade, seu salário e a quantidade de dependentes. Escreva um método que determina se o funcionário tem direito ao benefício, um segundo método para calcular o valor do benefício e um terceiro método que determina o novo valor do seu salário.

### Exercício 1.7. (OperacoesVetores)

Escreva um algoritmo que leia os valores de dois vetores de inteiros de 10 posições. Após a leitura, deverá ser criado um terceiro vetor para armazenar a soma dos elementos. Isto é, o valor da posição 0 do terceiro vetor corresponde à soma dos valores armazenados na posição 0 dos dois primeiros vetores. Neste sentido, crie vetores adicionais para armazenar a subtração, multiplicação e divisão (real) dos dois vetores iniciais.

### Exercício 1.8. (TrocaVetores)

Faça um programa que leia os valores de um vetor de  $n$  posições ( $n$  deverá ser um número par e será informado no início da execução do algoritmo). Após isso, troque os elementos das primeiras  $n/2$  posições com os elementos das últimas  $n/2$  posições, apresentando o resultado em tela. Finalmente, apresente a soma dos números pares e o produto dos números ímpares armazenados no vetor.

### Exercício 1.9. (OperacoesMatriz)

Escreva um algoritmo que faça a leitura de uma matriz quadrada de tamanho  $n$  (o tamanho também deve ser fornecido pelo usuário) com número inteiros. Após a leitura, o algoritmo deverá apresentar a soma dos valores abaixo da diagonal principal, a soma dos valores acima da diagonal principal e o produto dos valores da diagonal principal.

### Exercício 1.10. (GeraMatriz)

Faça um algoritmo que leia uma matriz de ordem  $N \times M$  com números inteiros e some cada uma das linhas, armazenando o resultado das somas em um vetor. A seguir, multiplique cada elemento da matriz pela soma da linha e mostre a matriz resultante.

### Exercício 1.11. (Palindromo)

Crie um programa que faça a leitura de uma palavra e verifique se a mesma é um palíndromo. Um palíndromo é uma palavra que, quando invertida, não é modificada. Exemplo: REVER.

**Exercício 1.12. (BuscaPalavra)**

Escreva um algoritmo que faça a leitura de um texto informado pelo usuário. Após isso, leia uma palavra também informada pelo usuário. O algoritmo deve determinar se a palavra encontra-se no texto informado. Utilize apenas a função `charAt(i)`, da classe `String`.

## 2. Conceitos básicos de orientação a objetos

### Exercício 2.1. (CadLivro)

Crie um programa que implemente a classe **Livro**. Esta classe deve conter o título do livro, nome do autor, editora e quantidade de páginas. Adicione um atributo que armazene a página atual (**paginaAtual**), para apresentação em um dispositivo eletrônico de leitura. Crie um método **virarPagina**, que incrementa o valor armazenado em **paginaAtual**. Após isso, crie uma segunda classe chamada **Main** (classe principal) que conterá o método **main**. Nesta classe, crie um objeto **Livro** e preencha seus atributos com valores lidos do usuário. Após isso, chame o método para virar uma página e apresente o objeto (seu estado) em tela. A classe **Livro** é apresentada abaixo.

Livro
<ul style="list-style-type: none"><li>– titulo: String</li><li>– autor: String</li><li>– editora: String</li><li>– numPags: int</li><li>– pagAtual: int = 0</li></ul>
<ul style="list-style-type: none"><li>+ métodos construtores</li><li>+ métodos set() e get()</li><li>+ virarPagina(): void</li></ul>

### Exercício 2.2. (ListaLivros)

Usando a classe **Livro**, criada no exercício anterior, crie uma lista para armazenar diferentes livros. Esta lista pode ser criada usando a coleção **ArrayList**. Crie um conjunto de objetos, solicite ao usuário o valor dos seus atributos e armazene-os na lista. Após isso, apresente o título e o número de páginas de cada livro armazenado. Utilize uma lista global e diferentes métodos para a criação dos objetos e sua apresentação.

### Exercício 2.3. (ConsultaLivro)

Com base no exercício anterior, crie um método para a consulta de livros. O usuário informa o título do livro desejado, o sistema faz a busca na lista de livros e apresenta seus dados, caso o encontre. Caso contrário, o sistema deve apresentar a mensagem "*Livro não encontrado*".

### Exercício 2.4. (ExcluiLivro)

Com base no exercício anterior, crie um método para exclusão de livros. O usuário informa o título do livro que deseja excluir, o sistema faz a busca do livro e o remove da lista, caso o encontre. Caso contrário, o sistema deve apresentar a mensagem "*Livro não encontrado*".

### Exercício 2.5. (AlteraLivro)

Com base no exercício anterior, crie um método para alteração de livros. O usuário informa o título do livro que deseja alterar, o sistema faz a busca do livro e, caso o encontre, solicita as novas informações ao usuário, atualizando seus campos. Caso contrário, o sistema deve apresentar a mensagem "*Livro não encontrado*".

### Exercício 2.6. (LivrosCompleto)

Com base nos métodos criados nos exercícios anteriores, crie um programa que apresente

ao usuário um menu com todas as opções (cadastro de livro, alteração, exclusão, consulta por título, consulta completa e sair). O usuário pode selecionar as opções desejadas e, ao terminar, seleciona a opção **sair**, que finaliza a execução do programa.

### Exercício 2.7. (Funcionario)

Crie um programa que implemente a classe **Funcionario** apresentada abaixo. Na classe principal da aplicação, deverá ser criada uma lista para armazenar os funcionários. Crie um menu que forneça ao usuário as seguintes operações:

Funcionario
<ul style="list-style-type: none"><li>- cpf: String</li><li>- nome: String</li><li>- idade: int</li><li>- salario: double</li><li>- tempoServico: int</li><li>- dependentes: int</li></ul>
+ métodos set() e get()

1. Incluir funcionário.
2. Excluir funcionário.
3. Alterar dados de um funcionário.
4. Consultar funcionário pelo CPF.
5. Consultar funcionários por tempo mínimo de serviço.
6. Consultar o salário médio dos funcionários cadastrados.
7. Consultar o total de dependentes dos funcionários cadastrados.

### Exercício 2.8. (Farmacia)

Crie um programa para gerenciamento de produtos de uma farmácia. Serão implementadas duas classes: **Medicamento** e **Cosmetico**. Um medicamento possui uma descrição, dosagem, nome do laboratório que o fabrica e seu preço. Um cosmético possui uma descrição, marca, número de lote e preço. Crie as representações dessas classes utilizando UML. No programa, crie métodos para o cadastro de medicamentos e cosméticos, bem como a consulta dos registros cadastrados. Crie um método que liste todas as marcas de cosméticos e a quantidade de cosméticos de cada uma delas. Crie um método para mostrar os medicamentos com preço maior que a média de preços dos cosméticos.

### Exercício 2.9. (Calculadora)

Desenvolva uma calculadora capaz de resolver três cálculos distintos: raízes de funções quadráticas pela fórmula de Bhaskara, hipotenusa de um triângulo retângulo pelo teorema de Pitágoras e área de trapézio. Crie uma classe para cada cálculo, definindo seus atributos em função dos valores necessários para o cálculo.

### 3. Associações simples

#### Exercício 3.1. (EquipeTreinador)

Considere as classes **Equipe** e **Treinador**. Uma equipe possui um treinador, e um treinador treina uma equipe. A equipe é composta por um nome (**String**) e uma categoria (**String**). O treinador possui um nome (**String**), número de registro (**int**) e um salário (**double**). Implemente estas classes com a respectiva associação (considere a navegabilidade de **Equipe** para **Treinador**), crie dois objetos (**Equipe** e **Treinador**) lendo seus valores do usuário e associe-os. Após isso, mostre o nome da equipe, sua categoria e o nome do seu treinador (o treinador deve ser acessado a partir da equipe, ou seja, usando a associação).

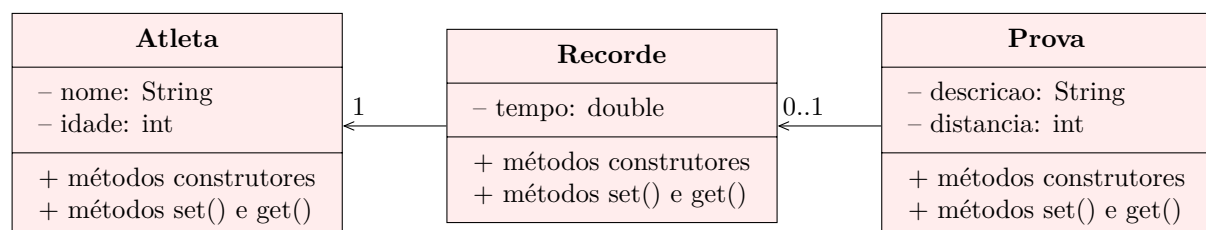
#### Exercício 3.2. (ListaEquipes)

Modifique o exercício anterior e torne o relacionamento bidirecional. Crie uma lista de equipes para armazenar os registros. O usuário poderá cadastrar uma série de equipes com seus respectivos treinadores. Ao final, mostre todas as equipes cadastradas juntamente com o nome do treinador e seu número de registro.

#### Exercício 3.3. (Natacao)

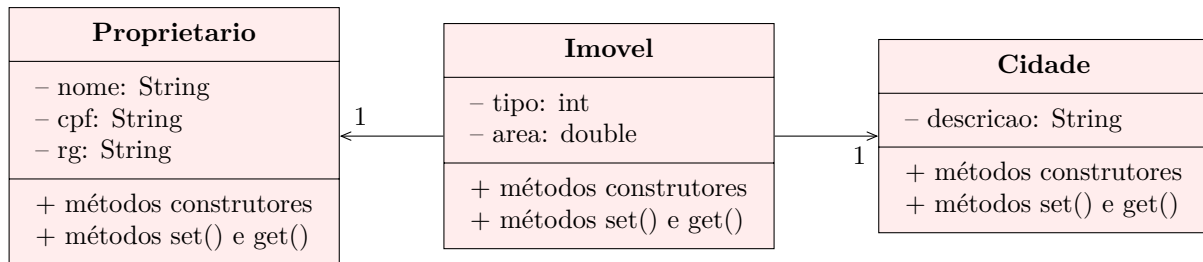
Considere um sistema para gerenciamento de provas de natação. O sistema deve controlar as provas e os respectivos recordes, armazenando o registro do atleta que o atingiu. Implemente a estrutura de classes abaixo e métodos para cadastro de atletas, provas e recordes. No cadastro do recorde, o programa deve apresentar as provas e solicitar ao usuário que selecione a opção desejada. Após selecionada a prova, o usuário deverá selecionar o atleta seguindo a mesma lógica anterior. Finalmente, o recorde é cadastrado, substituindo o recorde anterior, quando existente.

**OBS:** Uma prova não precisa, necessariamente, possuir um recorde vinculado. No entanto, o recorde precisa, necessariamente, possuir um atleta vinculado.



#### Exercício 3.4. (Imobiliaria)

Crie uma aplicação para o cadastro e manutenção de imóveis. O diagrama de classes abaixo mostra a estrutura da aplicação. Além dos seus atributos primitivos, um imóvel possui um proprietário, que também é modelado por uma classe. O imóvel também possui uma cidade, que também é modelada por uma classe. Note que o imóvel precisa de um proprietário e de uma cidade para ser cadastrado (dadas as multiplicidades das classes). Crie métodos para inserir registros fixos de proprietários e cidades. Crie um método para o cadastro de imóveis onde o usuário informa os dados do imóvel e seleciona o proprietário e a cidade. Crie métodos para a consulta de imóveis por tipo e consulta de imóveis de uma determinada cidade (os filtros são informados pelo usuário).



### Exercício 3.5. (Filmes)

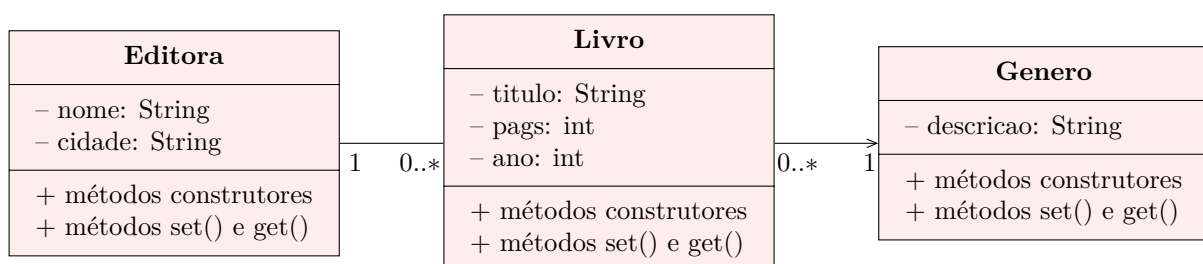
Considere um sistema para controlar a localização de filmes de uma locadora. Neste sentido, o sistema deve implementar a classe **Estante** (`numSala`, `numCorredor`) e **Filme** (`titulo`, `sinopse` e `duracao`). Um filme estará em uma estante, e uma estante pode conter zero ou muitos filmes. Implemente estas duas classes (com navegabilidade de **Filme** para **Estante**), solicite ao usuário as informações de uma estante com três filmes e de uma segunda estante com dois filmes. Faça a associação dos objetos e apresente-os em tela.

### Exercício 3.6. (ListaFilmes)

Modifique as classes do exercício anterior e inverta a navegabilidade da associação. Crie uma lista para armazenar as estantes e solicite ao usuário o cadastro de várias estantes com seus filmes. Ao final, mostre todas as estantes armazenadas na lista, juntamente com o título dos filmes da mesma.

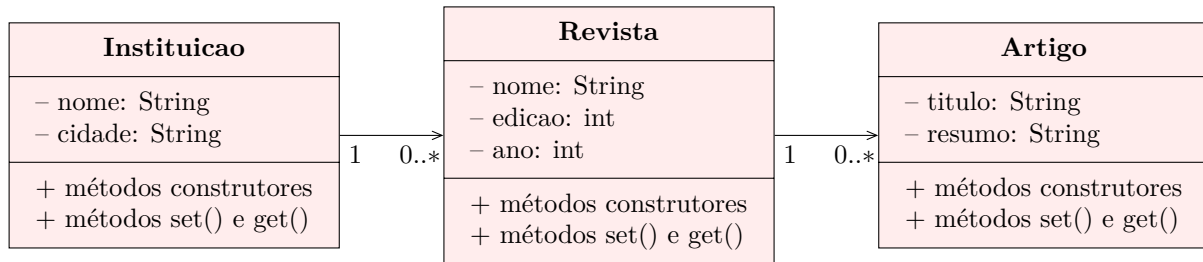
### Exercício 3.7. (Livros)

Crie uma aplicação para o cadastro e manutenção de livros. O diagrama de classes abaixo mostra a estrutura da aplicação. Crie métodos para criar conjuntos de editoras e de gêneros, armazenando os registros em listas. Crie um menu onde o usuário pode escolher as opções de cadastrar, excluir e consultar livros. No cadastro, o usuário deve selecionar uma editora e um gênero para vincular ao livro. Na consulta todas as informações do livro, editora e gênero deverão ser apresentadas.



### Exercício 3.8. (Artigos)

Crie uma aplicação para o cadastro de artigos científicos. Cada artigo é publicado em uma revista e cada revista pertence a uma instituição. A aplicação deverá permitir o cadastro de instituições, revistas e artigos, bem como a consulta de todos os artigos cadastrados.

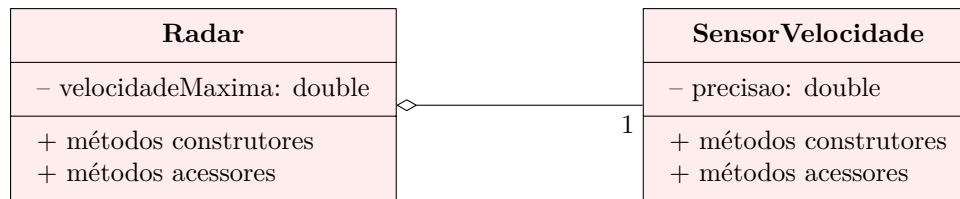




## 4. Agregação e composição

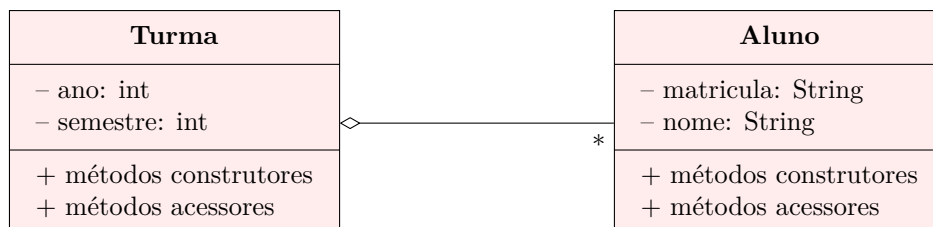
### Exercício 4.1. (RadarSensor)

Considere as entidades **Radar** e **SensorVelocidade**. Um radar possui um sensor de velocidade, que o informa da velocidade do veículo. O sensor de velocidade existe mesmo fora do relacionamento com o radar e pode estar relacionado a dois ou mais radares ao mesmo tempo. Logo, as duas entidades formam um relacionamento de agregação, conforme apresentado abaixo. Escreva o código para estas classes e implemente o método `List<Radar> radaresPrecisaoMinima(List<Radar> radares, double precisao)`. Este método recebe uma lista de radares e um valor de precisão, e deve retornar uma lista com os radares encontrados que possuem a precisão mínima exigida.



### Exercício 4.2. (TurmaAluno)

Considere as entidades **Turma** e **Aluno**. Os alunos são parte de uma turma, que é a entidade todo. Os alunos existem independente do relacionamento com a turma e um aluno pode pertencer a duas turmas simultaneamente. Logo, temos um relacionamento de agregação (veja diagrama abaixo). Escreva o código das classes e implemente o método `List<Turma> turmasDoAluno(List<Turma> turmas, String nomeAluno)`, que recebe uma lista de turmas e o nome de um aluno, retornando uma lista com as turmas das quais o aluno faz parte.



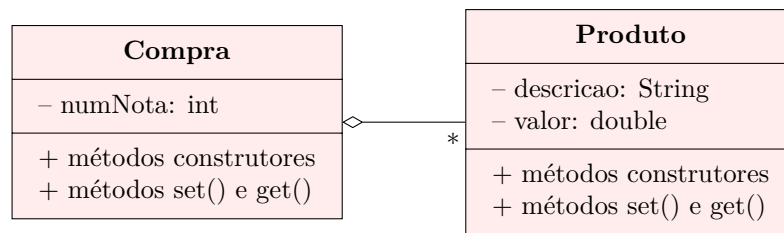
### Exercício 4.3. (Compras)

Considere um sistema para gerenciar compras feitas pela Internet. Cada compra possui o número da sua nota fiscal e agrega os produtos adquiridos, os quais possuem uma descrição e um valor. O diagrama de classes abaixo apresenta a estrutura dessas entidades. Implemente a estrutura de classes proposta e as funcionalidades de cadastrar uma compra, cadastrar os produtos da compra e listar o valor total de uma compra pelo número da sua nota fiscal.

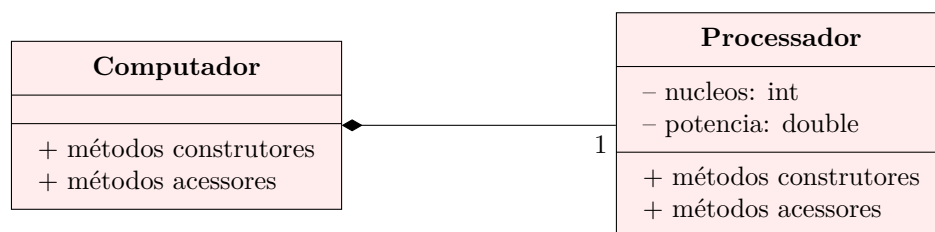
**OBS:** Mantenha uma lista de produtos na aplicação. Ao cadastrar um novo produto (sua descrição), a aplicação deverá verificar se este produto já está cadastrado. Neste caso, o produto já existente é vinculado à compra.

### Exercício 4.4. (ComputadorProcessador)

Considere as entidades **Computador** e **Processador**. Um computador possui um único

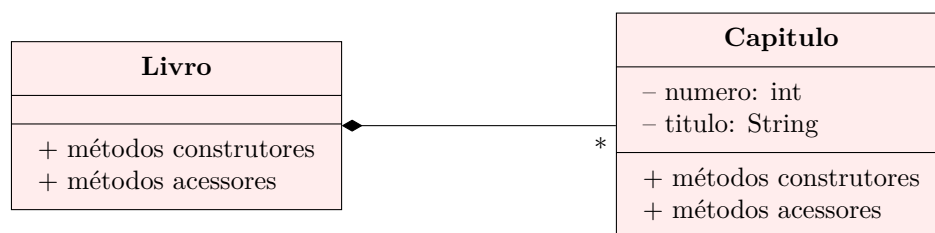


processador. O processador fica localizado dentro do computador e, portanto, não existe fora do relacionamento com o computador e não pode estar relacionado com dois ou mais computadores ao mesmo tempo (regras para definição de uma composição). Implemente as classes supracitadas e o método `mostraProcessadores(List<Computador> computadores)`. Com base na lista de computadores recebida, o método deve mostrar os processadores com mais de um núcleo.



#### Exercício 4.5. (LivroCapitulos)

Considere as entidades **Livro** e **Capitulo**. Um livro é composto por vários capítulos. O capítulo é parte de um livro e, portanto, não existe fora do relacionamento. Além disso, um capítulo não deve fazer parte de dois livros ao mesmo tempo. Logo, trata-se de uma composição. Apresente o código das classes supracitadas e a implementação do método `mostraCapitulos(Livro livro, String palavra)`, que recebe como argumento um livro e mostra os capítulos cujo título contenha a palavra buscada.



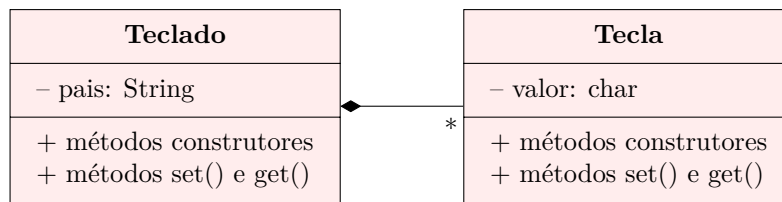
#### Exercício 4.6. (TecladoTecla)

Desenvolva uma aplicação para o armazenamento de teclados de diferentes países. Cada teclado é composto por um conjunto de teclas (diagrama de classes abaixo), as quais são identificadas pelo caracter que possuem. O usuário poderá incluir um novo teclado, juntamente com as suas teclas, bem como listar os teclados (e suas teclas) já cadastrados. Além disso, o usuário poderá informar uma palavra e o sistema deverá informar quais teclados são capazes de escrever a referida palavra (com base nas suas teclas).

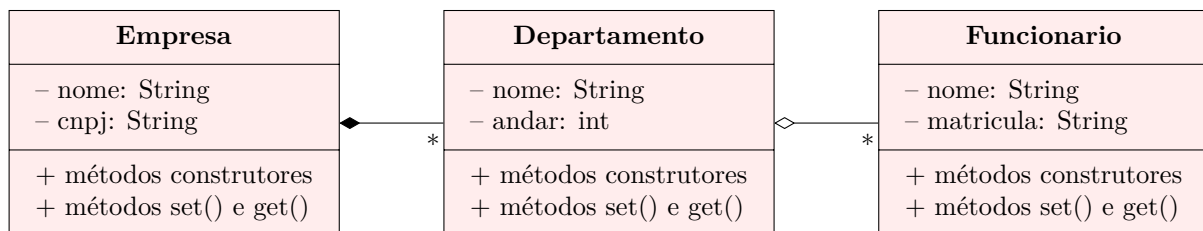
**OBS:** A aplicação não deve manter uma lista de teclas, pois uma tecla só existe em função de um teclado (conceito de composição).

#### Exercício 4.7. (GestaoEmpresas)

Crie uma aplicação para a gestão das empresas de um grupo. O diagrama de classes



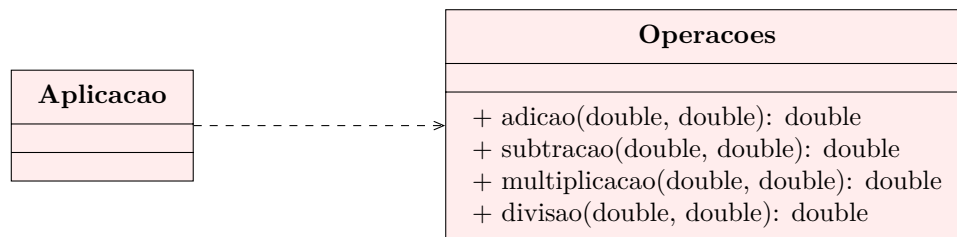
abaixo apresenta a estrutura das classes envolvidas. Uma empresa é composta por departamentos, os quais possuem funcionários. Um funcionário pode trabalhar em mais de um departamento ao mesmo tempo. Crie uma lista de empresas e uma lista de funcionários fixos. Implemente as opções de cadastro de departamento das empresas e a vinculação de funcionários aos seus departamentos.



## 5. Dependência

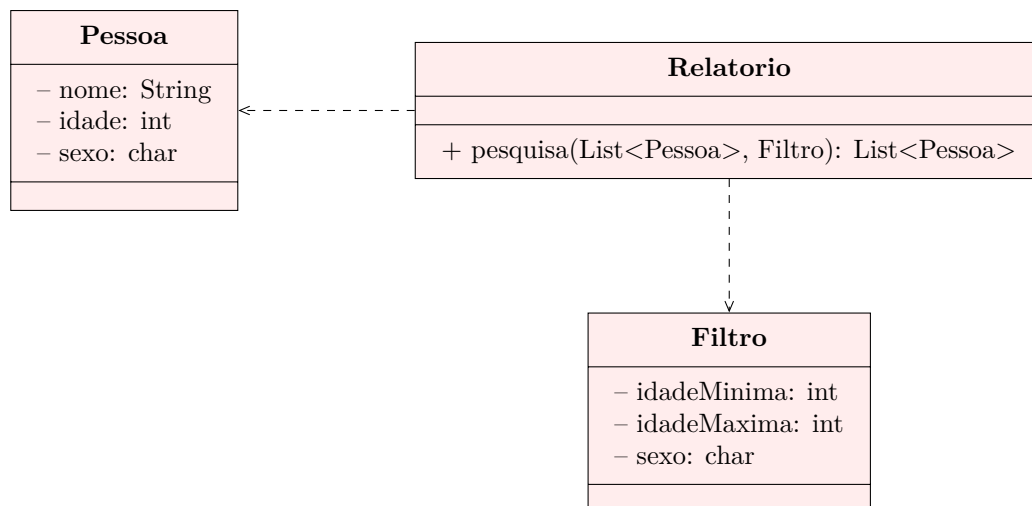
### Exercício 5.1. (Calculadora)

Desenvolva uma calculadora na qual o usuário informe, através de um campo de texto, a expressão matemática que deseja calcular. Considere expressões matemáticas simples, com dois valores e um operador aritmético simples (adição, subtração, multiplicação e divisão). Exemplos: “5 + 3”; “6 – 2”; “14 \* 3”; “15/3”. Após informada a expressão, o sistema deve analisar a entrada, realizar o cálculo correspondente e informar o resultado. O diagrama abaixo mostra a estrutura de classes para este exemplo. A classe **Aplicacao** faz a leitura dos dados e a apresentação dos resultados. A classe **Operacoes** é responsável por implementar as operações aritméticas suportadas e é utilizada pela classe **Aplicacao** por meio de uma dependência.



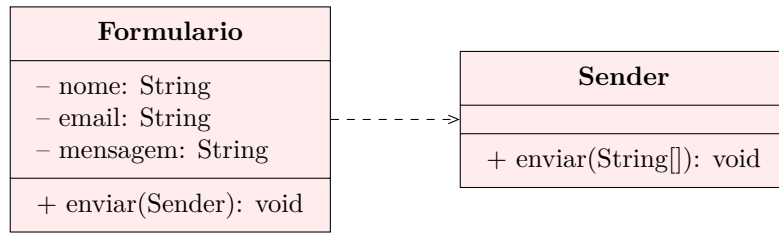
### Exercício 5.2. (RelatorioPessoas)

Crie um programa para a geração de relatórios de pessoas. Na classe **Aplicacao**, crie uma lista de pessoas com um conjunto inicial de registros. Um relatório recebe a lista de todas as pessoas cadastradas e devolve a lista filtrada de pessoas, a qual deve ser impressa em tela. O filtro pode ser feito por idade mínima e máxima e por sexo, sendo implementado por uma classe específica. O diagrama abaixo apresenta a estrutura de classes.



### Exercício 5.3. (EnvioFormulario)

Desenvolva um programa que leia do usuário os campos de entrada de um formulário e faça o envio destes dados (considere como envio a simples impressão dos dados em tela). O envio é feito por uma classe específica, da qual o formulário depende. O método de envio recebe um vetor de `String` com todos os dados e os imprime em tela. A estrutura de classes é apresentada abaixo.



## 6. Herança e polimorfismo

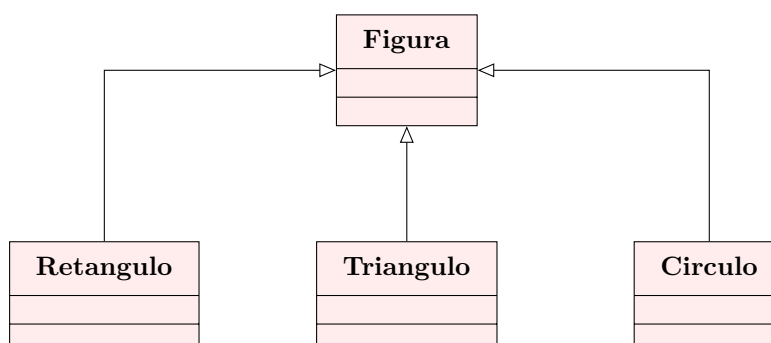
### Exercício 6.1. (ContasBancarias)

Considere um sistema que armazene contas bancárias. Uma conta possui o número de sua agência, número da conta e o saldo disponível. Além disso, esta entidade deve implementar um método de depósito que recebe uma quantia e adiciona ao seu saldo, bem como um método de saque que subtrai a quantia recebida do seu saldo, quando suficiente. Uma conta pode ser do tipo conta corrente ou conta poupança. A conta corrente possui um limite de crédito que pode ser utilizado quando o saldo não for suficiente para um saque. Isto é, o saldo pode ficar negativo até o limite de crédito da conta. Já a conta poupança possui uma taxa de juros que determina o rendimento mensal da mesma. Implemente a estrutura de classes descrita acima utilizando o conceito de herança. Implemente ainda uma classe Aplicacao que realiza as seguintes operações:

- Armazena uma lista de contas (que podem ser tanto contas correntes quanto poupanças).
- Método que cria registros aleatórios de contas e os armazena na lista.
- Método para sacar um valor determinado de todas as contas.
- Método que recebe duas contas e um valor e faz a transferência da primeira para a segunda conta.
- Método que atualiza o saldo das poupanças, adicionando o rendimento de acordo com sua taxa de juros.

### Exercício 6.2. (FigurasGeometricas)

Crie um sistema para armazenamento de figuras geométricas. Implemente a estrutura de classes abaixo, onde todas as figuras possuem uma cor. Todas elas devem implementar um método para cálculo da sua área e um método para cálculo do seu perímetro. Cada classe deve implementar os atributos necessários para executar estes cálculos. Crie uma classe Aplicacao que implemente um método que, dada uma lista de figuras geométricas, determine a área total de todas as figuras. Além disso, crie um método que, dada uma lista de figuras geométricas, apresente a cor da figura de menor perímetro.



### Exercício 6.3. (OperacoesCores)

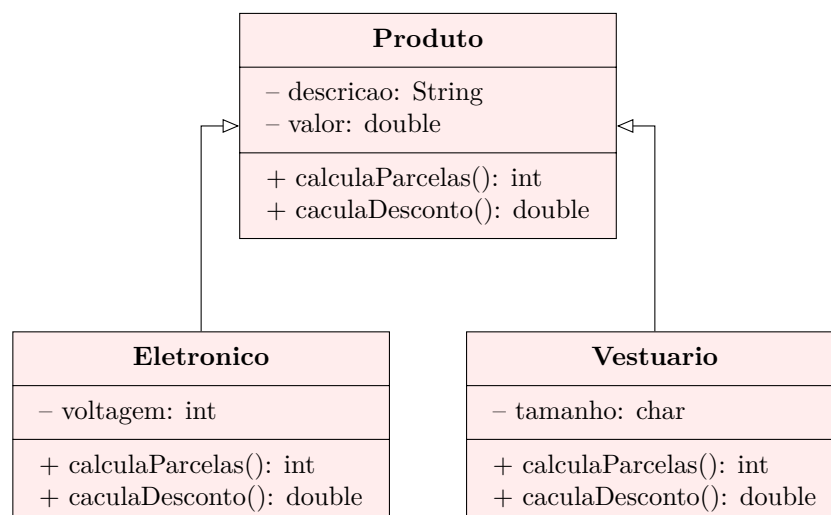
Utilizando as classes desenvolvidas no exercício anterior, crie uma classe Desenho, composta por figuras geométricas (composição). Crie um método (na classe Aplicacao) que receba um desenho e uma cor e mostre a área total da cor no desenho. Crie um segundo

método que receba um desenho e mostre as cores que o compõem e a respectiva área que cada cor ocupa no desenho. Finalmente, crie um terceiro método que receba uma lista de desenhos e uma cor e apresente os desenhos que não possuem a referida cor.

#### Exercício 6.4. (Loja)

Considere uma loja que vende equipamentos eletrônicos e roupas. Desenvolva um sistema para o gerenciamento dos produtos da loja. Cada produto mantém seu nome e seu valor. Eletrônicos possuem uma voltagem (110 ou 220) e itens do vestuário possuem um tamanho (P, M ou G). Cada produto possui um método para cálculo do número de parcelas, que por padrão é igual a 4x. No caso dos eletrônicos, produtos acima de R\$ 1000,00 podem ser parcelados em 10x. No caso de roupas, mais duas parcelas podem ser adicionadas ao parcelamento padrão, caso a roupa custe mais que R\$ 200,00. Além disso, os produtos implementam um método que calcula o desconto a ser concedido na venda. Por padrão, é concedido 5% de desconto, valor que aumenta para 10% nos eletrônicos. No caso das roupas, um desconto adicional de R\$ 15,00 é dado nas vendas. O sistema deve armazenar em uma lista única todos os produtos do estoque. Com base nisso, implemente os seguintes métodos:

- Cadastro de produtos;
- Venda de produtos (usuário seleciona um produto da lista e o sistema calcula o valor final e número de parcelas);
- Método que mostra todos os produtos da lista com desconto superior a R\$ 80,00;
- Método que mostra todos os produtos da lista que podem ser parcelados em mais de 5x;
- Método que apresente, para todos os produtos do estoque, seu valor de venda com e sem desconto. Ao final, o método deve mostrar os valores totais caso todos os produtos fossem vendidos com e sem desconto.







# Referências Bibliográficas

Anselmo, F. (2005). *Aplicando lógica orientada a objeto em Java*. Florianópolis: Visual-Books.

CAELUM (2017). *Apostila Java e Orientação a Objetos*. Curso FJ-11.

Deitel, H. (2010). *Java: Como programar. 8ª Edição*. Prentice Hall, Porto Alegre.

Larman, C. (2000). *Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos*. Bookman.

Santos, R. (2013). *Introdução à programação orientada a objetos usando java 2a edição*. Elsevier Brasil.

Schildt, H. (2013). *Java: A Referência Completa*. Alta Books.