Desenvolvimento de sistemas

Linguagem de programação orientada a objetos

Como já foi citado anteriormente nos conteúdos desta unidade curricular quanto aos conceitos iniciais da POO (programação orientada a objetos), é preciso ver na prática a sua utilização por meio de uma linguagem de programação. Neste momento, a linguagem utilizada para exemplificar os conceitos do paradigma orientado a objetos será o Java, que, apesar de ser uma linguagem multiparadigma (pode ser usada em outros paradigmas da programação, como, por exemplo, o estruturado), é muito poderosa quando utilizada na POO, apresentando diversas vantagens como reutilização do código, facilidade de manutenção e atualização, além de grande capacidade de escalabilidade.

Definição de classes no Java

ALUNO
NOME
MATRÍCULA
CURSO

Tabela 1 – Classe **alunos**, que será implementada no Java
Fonte: Senac EAD (2022)

Você estudou anteriormente que classe é uma abstração, um molde de qualquer elemento do mundo real. Nesta seção do conteúdo, você aprenderá como essa relação acontece dentro de uma linguagem de programação e quais são suas melhores formas de utilização para resolução de problemas do mundo real.

Neste exemplo, semelhante ao visto no conteúdo **Orientação a objetos** desta unidade curricular, você tem uma classe chamada **Aluno**, que contém como atributos os elementos **Nome do Aluno**, **Número da Matrícula** e **Curso**.

Sem ainda se preocupar com métodos e outras nomenclaturas dentro da linguagem, conheça mais agora sobre a criação de uma classe, apenas considerando os atributos desta.

A seguir, confira uma ideia inicial da criação de classes, já com linhas de código Java:

```
public class Aluno {
    String nome;
    int matricula;
    String cpf;
}
```

Não se preocupe neste momento com palavras como *public* e *private*, que serão posteriormente explicadas no item sobre encapsulamento, deste conteúdo.

No código citado, foi criada uma classe **Aluno**, por meio da palavra reservada “class” e como atributos estão presentes “nome”, primeiro atributo que é do tipo *string* (variável tipo texto), um segundo atributo “matricula”, um INT (variável do tipo inteiro, números sem vírgula) e, como terceiro atributo, há “cpf”, que é outra *string*.

É possível analisar outras possibilidades de criação de classes para um melhor entendimento desta etapa, como mostrado no exemplo a seguir:

ANIMAL
NOME
RAÇA
PESO

Tabela 2 – Classe animal que será implementada no Java
Fonte: Senac EAD (2022)

Agora será criado o código das classes **Animal**, de acordo com o esquematizado anteriormente.

```
public class Animal {
    String nome;
    String raca;
    float peso;
}
```

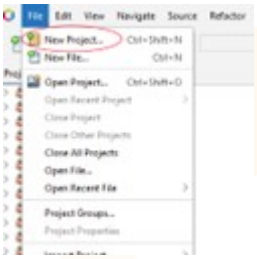
Nesse código, há uma classe **Animal**, e, como atributos, foram utilizados “nome”, que é uma *string*; como segundo atributo o item “raca” (sem “ç”, que não se utiliza em nomes de atributos e variáveis), que também é uma *string*; e, como terceiro atributo, o item “peso”, que é do tipo *float* (número que aceita vírgula).

No caso citado, poderia ainda ser utilizado o tipo *double* para o atributo “peso”, porém ele utilizaria mais memória que o *float*, que utiliza menos casas decimais.

Antes do seu primeiro desafio, entenda como é criado o primeiro projeto no NetBeans.

Passo 1

Com o programa NetBeans aberto, clique em File e depois em **New Project**, ou utilize o atalho **Ctrl + Shift + N**, como demostrado na imagem a seguir:



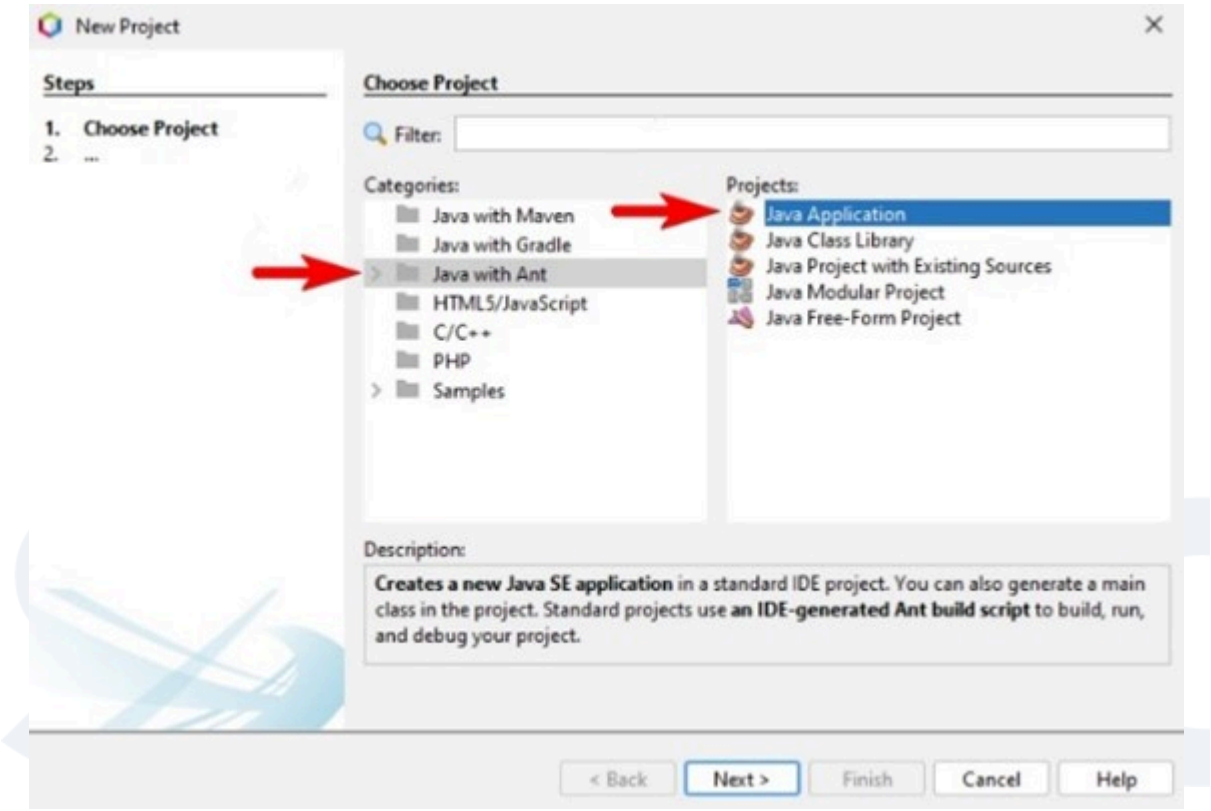
(objetos/fig1.jpg)

Figura 1 – Menu de criação de um novo projeto no NetBeans 13
Fonte: NetBeans 13 (c2017-2020)

Imagem da criação de um projeto no NetBeans, que mostra o momento do clique no menu “File” no canto superior esquerdo da tela, em que se abre uma caixa com diversos itens, entre os quais foi escolhido o item “New Project”, referenciado com uma elipse vermelha ao seu redor para facilitar a visualização.

Passo 2

Depois de clicar no item **New Project**, uma nova janela será aberta, na qual você deve escolher o tipo de projeto desejado. Nos exemplos deste material, serão utilizados sempre a categoria **Java with Ant** e o tipo de projeto **Java Application**:



(objetos/fig2.jpg)

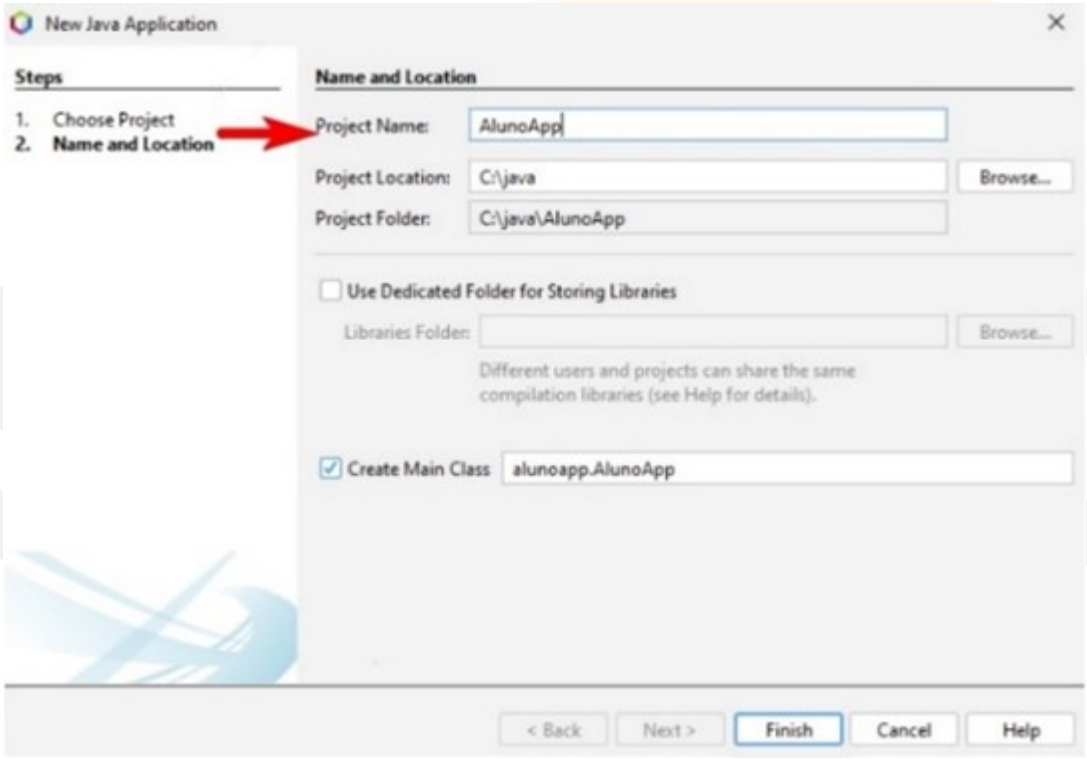
Figura 2 – Tela de escolha da categoria e tipo de projeto
Fonte: NetBeans 13 (c2017-2020)

Depois de clicar no item anterior chamado “New Project”, ou “novo projeto”, em português, será aberta uma caixa/janela com fundo cinza, na qual será preciso escolher a categoria e o projeto escolhido. Dentro dessa caixa há duas caixas menores de fundo branco, em que, na primeira, escolhe-se o item “Java with Ant”, no qual se utiliza uma seta vermelha para demonstrar o local que foi clicado, e, na segunda caixa, escolhe-se o item “Java Aplication”, também apontado por uma seta vermelha para melhor visualização. Após escolhidos, clica-se no botão “Next”, que aparece abaixo, na tela, com uma borda azul e é o primeiro dos botões que não está com transparência (não habilitado).

Depois de clicar na categoria **Java with Ant** e no tipo de projeto **Java Application**, clique no botão **Next**.

Passo 3

Depois de clicar em **Next**, aparecerá uma nova janela, na qual será possível definir nome e localização do projeto. Neste primeiro momento, preocupe-se apenas com o nome do projeto:



(objetos/fig3.jpg)

Figura 3 – Tela de escolha do nome e localização do projeto.
Fonte: NetBeans 13 (c2017-2020)

Nova janela de fundo cinza, na qual será escolhido o nome do projeto. Há duas partes internas, uma caixa de fundo branco a esquerda e uma de fundo cinza à direita. Na caixa de fundo cinza há um campo, no qual está digitado o nome do projeto “AlunoApp”, indicado por uma seta vermelha para permitir uma melhor visualização. Abaixo estão botões para serem clicados e um chamado “Finish”, que aparece com uma borda azul e é o primeiro dos botões que não está com transparência (não habilitado).

Com os três primeiros passos concluídos, tem-se um arquivo “AlunoApp.java”, criado com o mesmo nome da classe principal, local onde aparece o método **main** (**public static void main(String[] args)**), e, a partir deste momento, é possível criar

quantas classes forem necessárias para a continuidade do trabalho. Isso será demonstrado com a criação da classe **aluno**, utilizada durante este material de estudo por meio dos passos a seguir:

Passo 4

Com a classe principal criada e inicializada na tela, clique com o botão direito em cima do pacote “alunoapp”, escolha o item **new** e depois, o subitem **Java Class**.

Em Java, pacote, ou *package*, é o local onde ficam agrupadas as classes que se relacionam no projeto em uma mesma estrutura hierárquica, ou seja, as classes que do projeto “AlunoApp”.

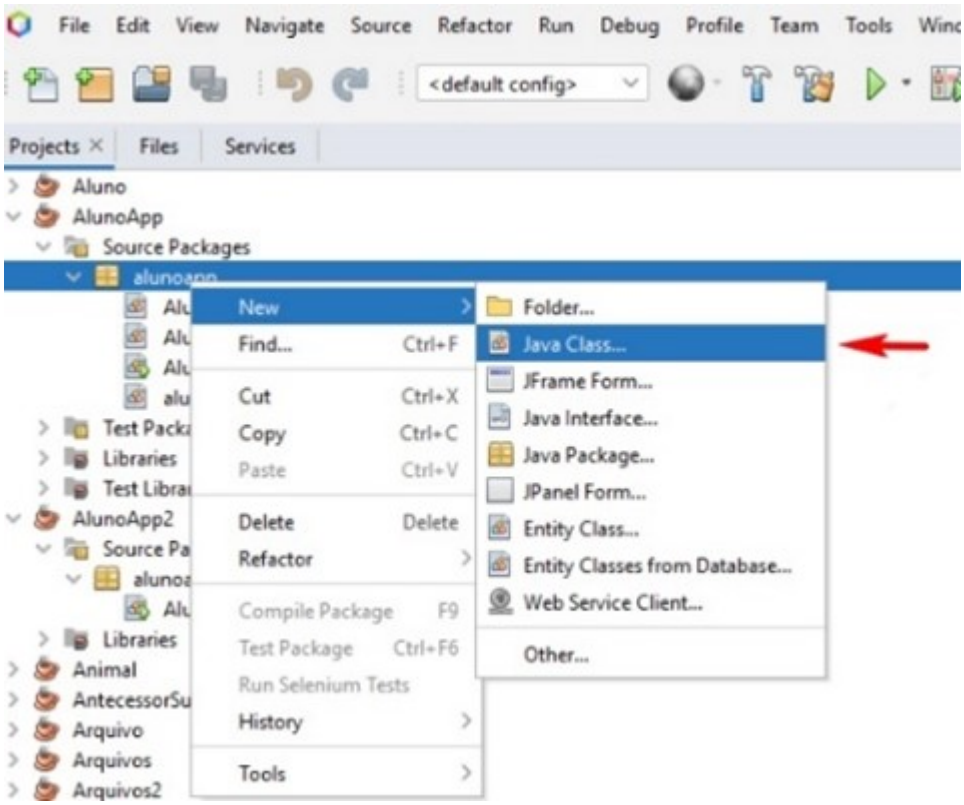


Figura 4 – Menu de criação de uma nova classe Java no NetBeans

Fonte: NetBeans 13 (c2017-2020)

Imagem da criação de uma nova classe Java no NetBeans, que mostra o momento do clique com o botão direito do mouse no pacote “alunoapp”, que abre duas caixas cinzas que representam os submenus. Na primeira, clica-se no item “new” no canto superior esquerdo da tela e, segunda caixa, no item “Java Class”, indicado por uma seta vermelha para facilitar a visualização.

Passo 5

Depois de clicar em **Java Class**, aparecerá uma nova janela na qual se pode definir nome e localização da classe que será criada. Neste primeiro momento, preocupe-se apenas com o nome da classe.

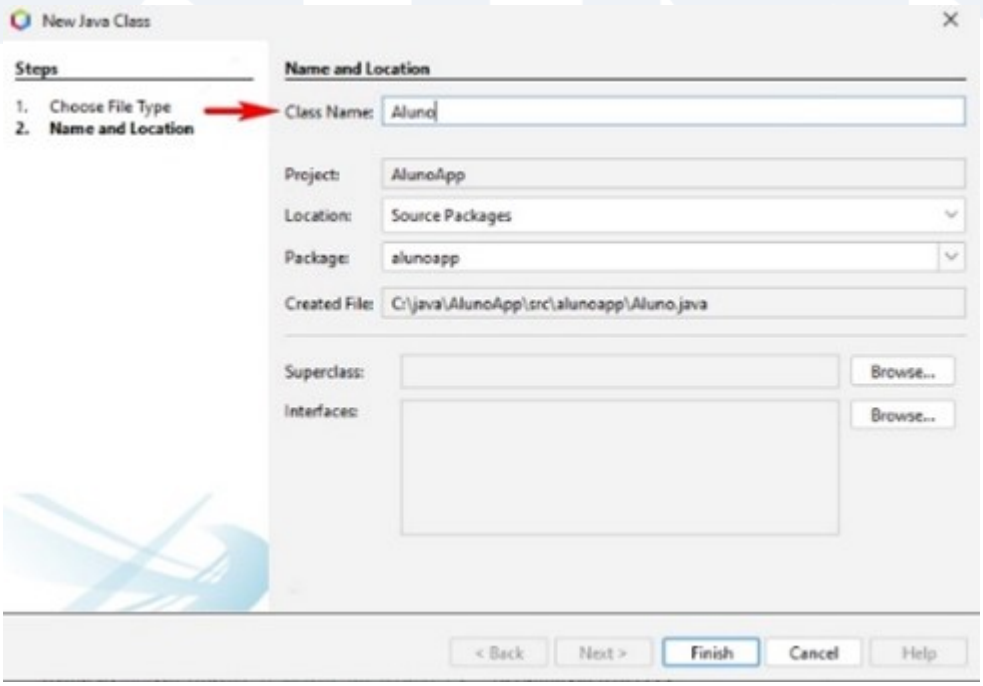


Figura 5 – Menu de criação de uma nova classe Java no NetBeans
Fonte: NetBeans 13 (c2017-2020)

Imagem da criação de uma nova classe Java no NetBeans que mostra o momento do clique com o botão direito do mouse no pacote “alunoapp”, que abre duas caixas cinzas que representam os submenus. Na primeira, clica-se no item

“new” no canto superior esquerdo da tela e, na segunda caixa, no item “Java Class”, indicado por uma seta vermelha para facilitar a visualização.

Agora, abrindo o arquivo “Aluno.java”, já é possível incluir os dados da classe descrita no início desse exemplo. O arquivo, neste momento, já tem a estrutura básica para a classe:

```
package alunoapp;
/**
 *
 * @author
 */
public class Aluno {

}
```

Assim, resta apenas completá-lo com os atributos da classe **Aluno**.

Relembre o seguinte código:

```
public class Aluno {
    String nome;
    int matricula;
    String cpf;
}
```

Compile o programa. Ao executá-lo, nada diferente deve acontecer, pois não foi alterado o método **main()** da classe principal (em “AlunoApp.java”, neste caso).

Crie as classes **Carro**, **Celular**, **Professor** e **Música**, definindo quatro atributos para cada uma delas. Primeiramente, planeje as classes em uma folha de papel e depois passe para a ferramenta NetBeans 13, como mostrado nos exemplos anteriores.

Como utilizar classes e objetos no Java

Entende agora a importância do uso da POO e suas vantagens em relação ao paradigma estruturado de programação. Na linguagem estruturada, utiliza-se uma variável sempre que é necessário armazenar um valor durante o funcionamento do programa e, a cada novo armazenamento, é preciso criar novas variáveis. Na orientação a objetos, por sua vez, é possível criar uma classe e instanciá-la quantas vezes forem necessárias por meio de objetos, de seus atributos e de métodos, o que facilita muito o trabalho do programador quando ele entende como funcionam as particularidades desse tipo de programação

Relembre a primeira classe criada e entenda como ela poderia ser utilizada em um programa real.

```
public class Aluno {
    String nome;
    int matricula;
    String cpf;
}
```

A partir deste momento, você começará a criar (instanciar) seus primeiros objetos e ver na prática o maravilhoso mundo da orientação a objetos, no qual, por meio de um esqueleto (classe), será possível transformar a abstração em objetos reais. Veja o código a seguir e atente-se aos comentários. Transcreva-o ao NetBeans na classe principal (**AlunoApp**) do projeto criado anteriormente.

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

  Aluno aluno1 = new Aluno(); /* Aqui é instanciado o primeiro objeto chamado alu
no1, por meio do uso da palavra reservada new */

  aluno1.nome="Celso Medeiros";
/* Aqui é colocado o valor “Celso Morais” no atributo nome do aluno1 */

  aluno1.matricula=1;
/* Aqui é colocado o valor “1” no atributo matrícula do aluno1 */

  aluno1.cpf="00134578090";
/* Aqui é colocado o valor “0013457890” no atributo cpf do aluno1 */


/* Logo abaixo, será utilizado o comando System.out.println, que é usado para mos
trar uma informação na tela do usuário */

/* Comando para mostrar o nome do aluno1 */
  System.out.println("Nome do aluno: " + aluno1.nome);

/* Comando para mostrar a matrícula do aluno1 */
  System.out.println("Matrícula do aluno: " + aluno1.matricula);

/* Comando para mostrar o cpf do aluno1 */
  System.out.println("CPF do aluno: " + aluno1.cpf);
}
```

Depois de rodar esse código, você terá este resultado na tela:

Para rodar o código, você deve apertar o botão de atalho **F6** ou utilizar o ícone simbolizado com uma seta verde, como mostrado a seguir:

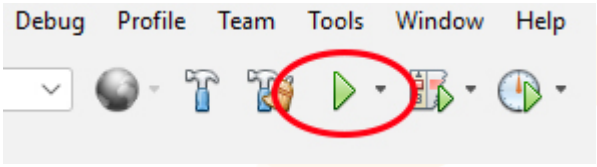


Figura 6 – Ícone para rodar o programa no NetBeans
Fonte: NetBeans 13 (c2017-2020)

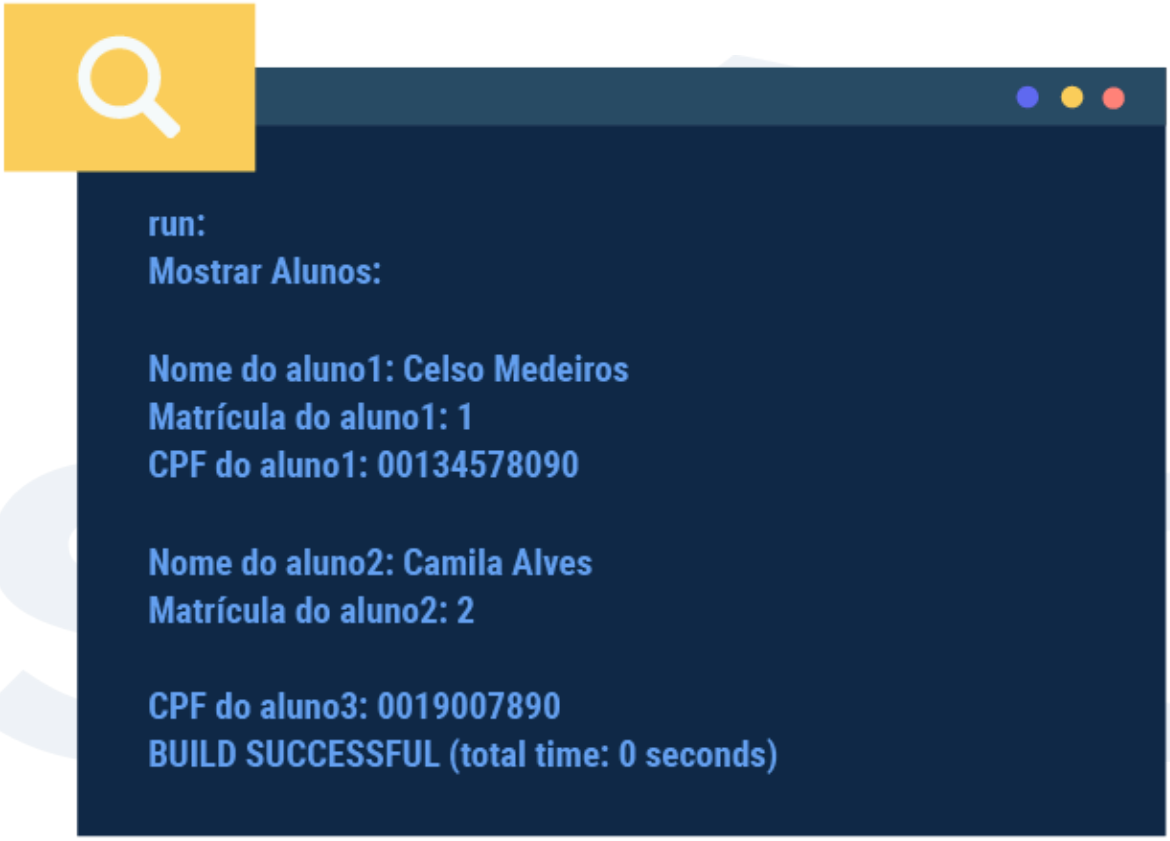
A imagem representa o ícone de uma seta verde na parte central, que é utilizado para rodar o programa no Java. Ele tem ao seu redor uma elipse vermelha para facilitar sua visualização.

Para melhorar, você criará novos objetos da mesma classe, mostrando que é possível iniciar todos os atributos de uma classe ou apenas os necessários para cada objeto. Por exemplo, é possível criar um objeto “aluno1” com nome, matrícula e CPF, um “aluno2” com nome e matrícula e um objeto “aluno3” somente com CPF, ou seja, você pode instanciar apenas o que precisa para cada um.

Nos exemplos a seguir, será instanciado mais de um objeto da mesma classe. Atente-se ao código e aos comentários contidos nele.

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

    Aluno aluno1 = new Aluno();
    Aluno aluno2 = new Aluno();
    Aluno aluno3 = new Aluno();
    /*Aqui são instanciados três objetos da mesma classe aluno, e agora é possíve
l decidir qual atributo se quer utilizar para cada um deles */

    aluno1.nome="Celso Medeiros";
    aluno1.matricula=1;
    aluno1.cpf="00134578090";

    aluno2.nome="Camila Alves";
    aluno2.matricula=2;

    aluno3.cpf="0019007890";

    /*Criados os objetos, deve-se mostrar os resultados na tela */

    System.out.println("Mostrar Alunos:/n"); /*o comando \n serve para pular uma
linha após o mostrar alunos */

    System.out.println("Nome do aluno1: " + aluno1.nome);
    System.out.println("Matrícula do aluno1: " + aluno1.matricula);
    System.out.println("CPF do aluno1: " + aluno1.cpf);

    System.out.println("/n"); /*o comando \n aqui também pula uma linha, mesmo co
locado sozinho dentro do comando System.out.println */

    System.out.println("Nome do aluno2: " + aluno2.nome);
    System.out.println("Matrícula do aluno2: " + aluno2.matricula);

    System.out.println("/n");
    System.out.println("CPF do aluno3: " + aluno3.cpf);

}
```

Após rodar esse código, você terá este resultado na tela:


run:
Mostrar Alunos:
Nome do aluno1: Celso Medeiros
Matrícula do aluno1: 1
CPF do aluno1: 00134578090
Nome do aluno2: Camila Alves
Matrícula do aluno2: 2
CPF do aluno3: 0019007890
BUILD SUCCESSFUL (total time: 0 seconds)

Métodos

Os métodos são uma das partes mais importantes de um objeto. Eles são o local no qual acontecem as ações do objeto e devem, entre outras coisas, ser utilizados para fazer cálculos, comparações, ordenações etc.

Um método pode ou não retornar um valor, dependendo de como foi configurado. A seguir, ficará demonstrada a forma de utilização dessa poderosa etapa da POO.

Para exemplificar o uso dos métodos, será resgatada a classe **Aluno**, mas agora com mais um atributo chamado “idade”, deixando a classe com a seguinte configuração:

ALUNO
NOME
MATRÍCULA
CURSO
IDADE

Tabela 3 – Classe Aluno, na qual serão criados os métodos
Fonte: Senac EAD (2022)

Funções

Antes de implementar o primeiro método, é importante relembrar e um conhecimento já aprendido durante o curso, que são as funções, pois elas têm algumas semelhanças importantes com os métodos na POO. Para fazer esta

comparação, analise a função **calcularMedia()**. Nesta função, você terá como passagem de parâmetro dois valores do tipo real.

A seguir, veja os códigos das funções em Portugol:

```
programa
{

    funcao inicio() // Área principal do programa onde é feita a chamada da funçã
o
    {
        // Neste caso, o resultado retornado por calcularMedia seria 3,5
        escreva("O resultado do primeiro cálculo é: ", calcularMedia (3.0, 4.0))

    }

    // Função em Portugol
    funcao real calcularMedia (real valor1, real valor2)
    {
        real resultado
        resultado = (a + b)/2
        retorne resultado
    }
}
```

É possível notar que a função somar recebe dois parâmetros do tipo real: o primeiro é chamado de “valor1” e o segundo é chamado de “valor2”. Dentro da função, é criada uma variável chamada de “resultado”, que recebe o cálculo da média entre as duas notas recebidas e retorna esse resultado ao programa.

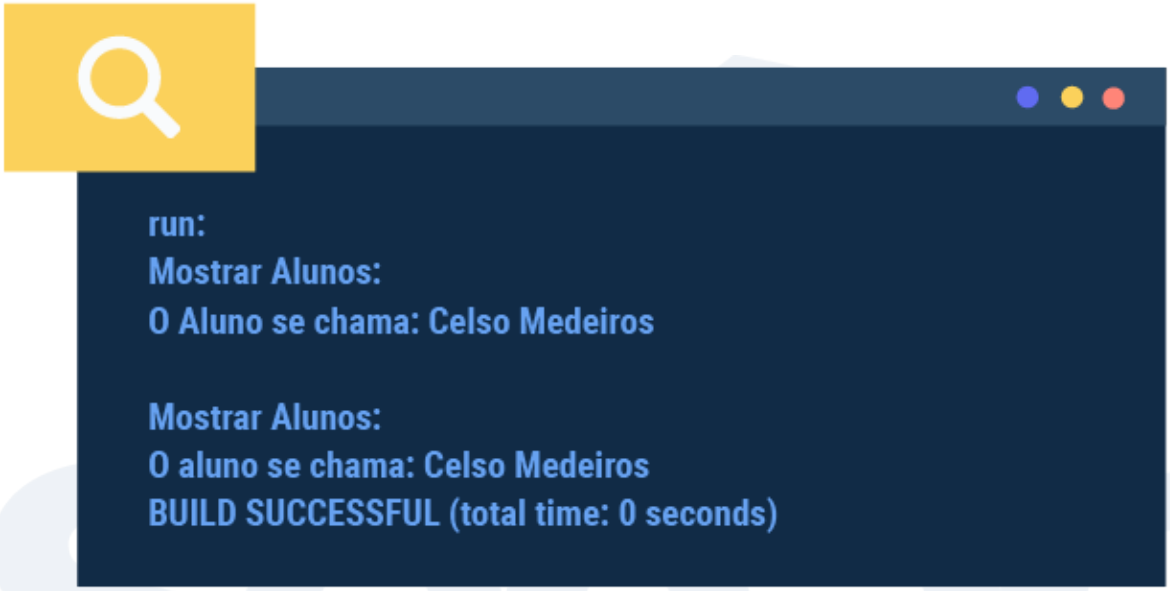
Essa forma de programação é muito semelhante com a que será vista nos métodos criados com passagem de parâmetro, que, da mesma forma, agiliza o código e o reutiliza. A diferença, porém, é que, nos métodos, muitas vezes o valor dos atributos dos objetos é modificado quando esta chamada é feita. Por exemplo, se o caso citado fosse um método e tivesse um atributo “resultado”, a partir deste momento o valor desse atributo poderia já se tornar “3,5”, resultado do exemplo mencionado.

Nas próximas linhas de códigos, você começará a implementação dos métodos e descobrirá as incríveis vantagens de sua utilização. Atente-se aos comentários.

```
public class Aluno {
    String nome;
    int matricula;
    String cpf;
    int idade;  /* Novo atributo do tipo int, inserido na classe Aluno */

  /* aqui, será criado o primeiro método do tipo void, ou seja, não retorna nenhu
m valor de variável*/
    public void mostrarNome(){
        System.out.println("O Aluno se chama: "+nome);
    }
}
```

```
public class AlunoApp {

    public static void main(String[] args) {

        Aluno aluno1 = new Aluno();


        aluno1.nome = "Celso Medeiros";

        System.out.println("Mostrar Alunos:");
        aluno1.mostrarNome(); /* chamando o método criado */
        System.out.println("\nMostrar Alunos:");
        System.out.println("O aluno se chama: " + aluno1.nome);
    }
}
```

O resultado da execução será algo como o exemplo a seguir:



Nos nomes dos métodos, será utilizado o padrão da linguagem chamado **CamelCase**, no qual se deixa em maiúscula a primeira letra de cada palavra (de um termo composto), com exceção do primeiro termo. Por exemplo, o método **mostrarmaiornota** fica **mostrarMaiorNota**, facilitando a leitura e o padrão de criação.

Analisando o código, note que as duas linhas de código mostradas em seguida retornaram o mesmo valor. Talvez você se pergunte, na primeira vez em que se deparar com um código desse, “por que utilizar métodos se posso simplesmente usar a segunda linha de código para mostrar um atributo de um objeto?”. A resposta torna-se mais simples a cada exemplo de utilização dos métodos analisado, mas saiba de antemão que, com o uso dos métodos, o código fica mais organizado, enxuto, reutilizável e seguro. Em seguida, note também que, cada vez que quiser chamar o nome de um aluno, você utilizará **aluno.mostrarNome()** e não precisará escrever a linha de código maior aprendida anteriormente.

```
aluno1.mostrarNome();

System.out.println("O aluno se chama: " + aluno1.nome);
/* Forma anterior sem o uso do método */
```

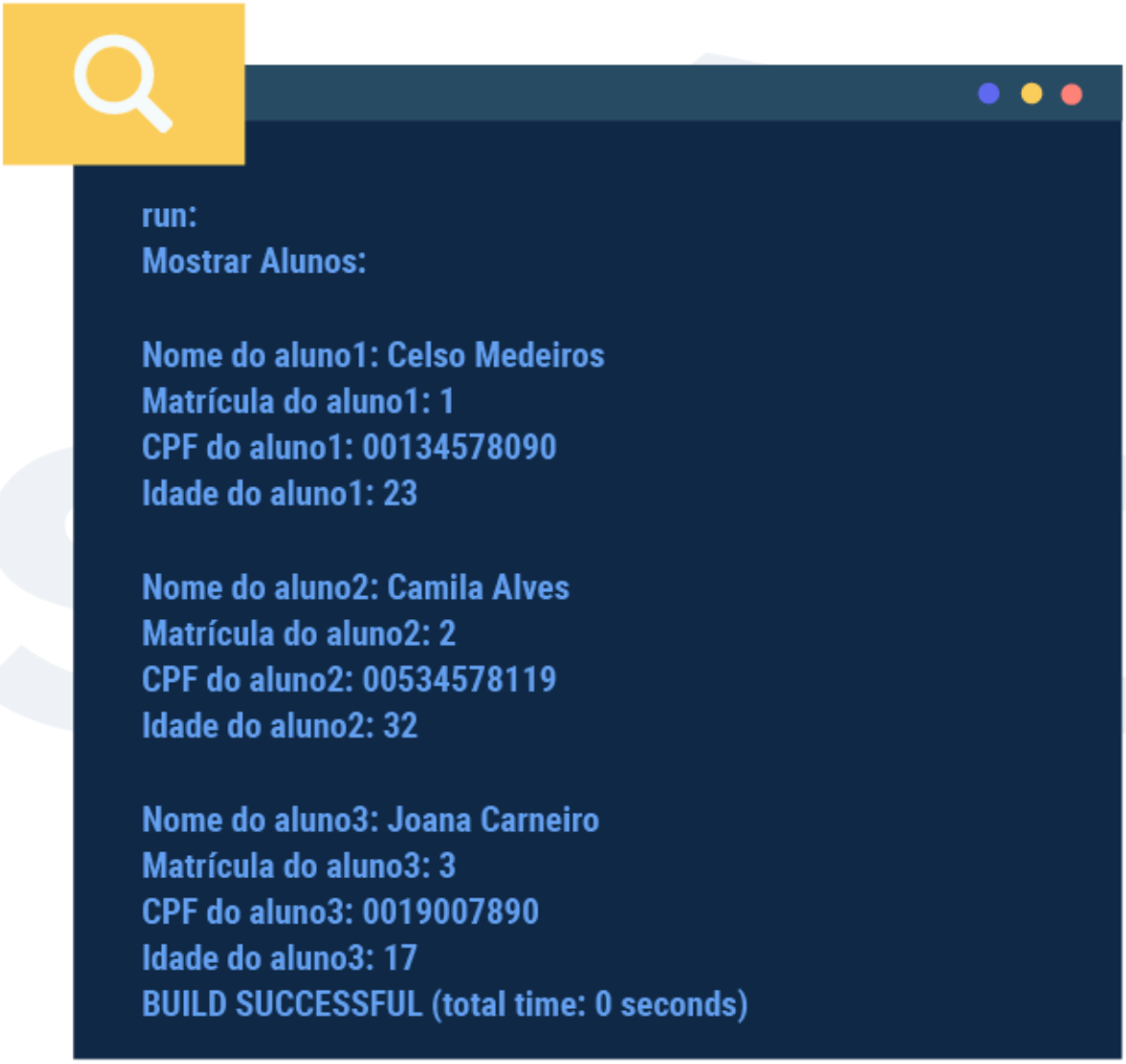
Utilizando o código exemplificado anteriormente, crie três métodos para mostrar matrícula, CPF e idade do aluno. Use como base o método **aluno1.nome** e sempre utilize verbos para os nomes dos métodos.

Pense em uma situação em que, querendo uma descrição completa de um aluno, você conseguisse com apenas uma linha de código apresentar o nome dele, sua matrícula, seu CPF e sua idade. Isso seria possível?

Comece instanciando alguns objetos e mostrando as informações completas deles, primeiro sem utilizar métodos e depois utilizando-os.

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

    Aluno aluno1 = new Aluno();
    Aluno aluno2 = new Aluno();
    Aluno aluno3 = new Aluno();


    aluno1.nome="Celso Medeiros";
    aluno1.matricula=1;
    aluno1.cpf="00134578090";
    aluno1.idade=23;

    aluno2.nome="Camila Alves";
    aluno2.matricula=2;
    aluno2.cpf="00534578119";
    aluno2.idade=32;


    aluno3.nome="Joana Carneiro";
    aluno3.matricula=3;
    aluno3.cpf="0019007890";
    aluno2.idade=17;

    System.out.println("Mostrar Alunos:/n");

    System.out.println("Nome do aluno1: " + aluno1.nome);
    System.out.println("Matrícula do aluno1: " + aluno1.matricula);
    System.out.println("CPF do aluno1: " + aluno1.cpf);
    System.out.println("Idade do aluno1: " + aluno1.idade);

    System.out.println("/n");
    System.out.println("Nome do aluno2: " + aluno2.nome);
    System.out.println("Matrícula do aluno2: " + aluno2.matricula);
    System.out.println("CPF do aluno2: " + aluno2.cpf);
    System.out.println("Idade do aluno2: " + aluno2.idade);

    System.out.println("/n");
    System.out.println("Nome do aluno3: " + aluno3.nome);
    System.out.println("Matrícula do aluno3: " + aluno3.matricula);
    System.out.println("CPF do aluno3: " + aluno3.cpf);
    System.out.println("Idade do aluno3: " + aluno3.idade);

}
```

O código retornará o seguinte resultado:

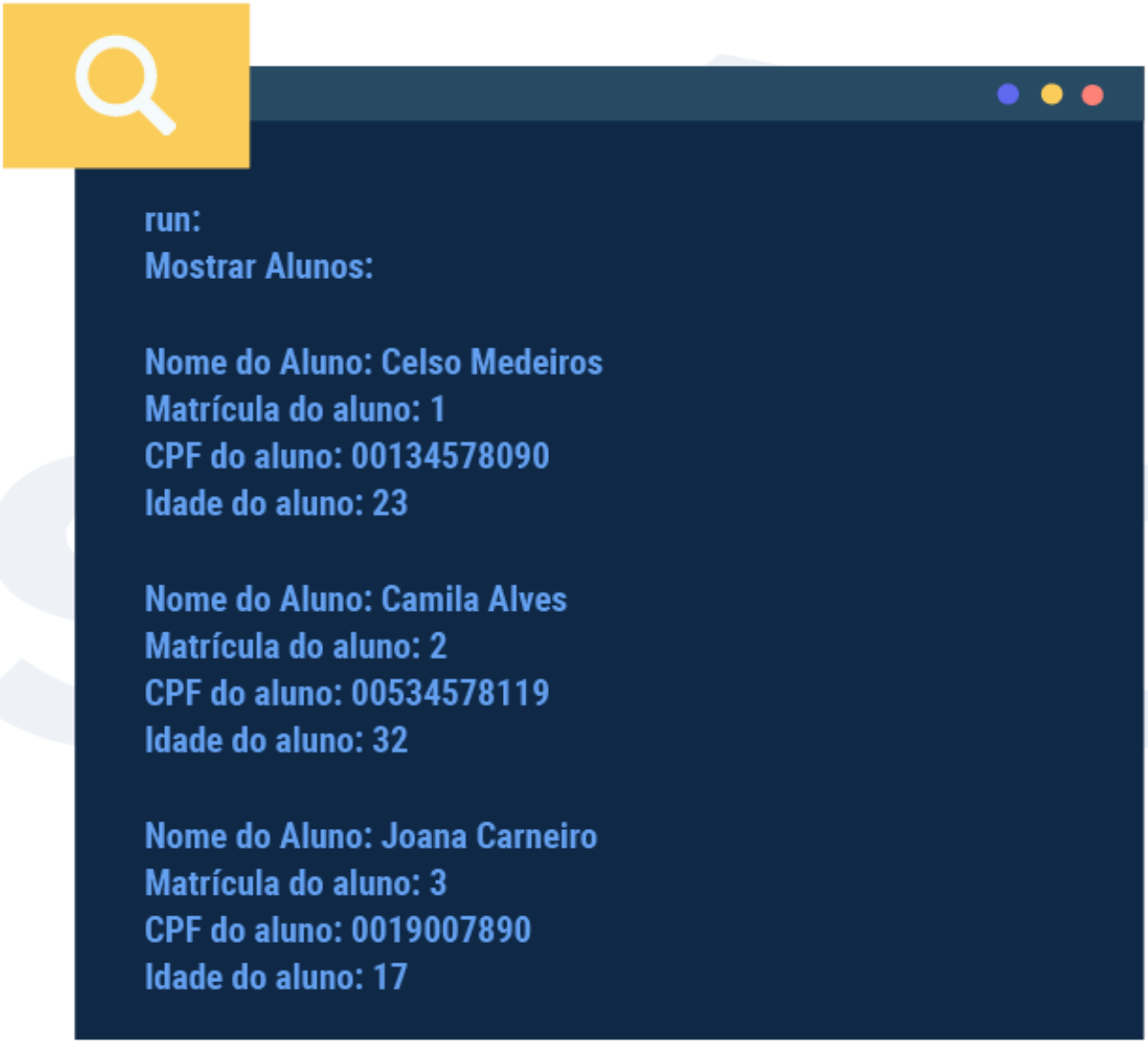


Observe que é preciso escrever um código bem grande a cada vez que se quer mostrar a informação completa de um objeto “aluno” criado. Isso poderia ter sido evitado se fosse criado um método para descrever um aluno e esse método fosse chamado sempre que necessário, como será demonstrado substituindo o método **mostrarNome()** criado no exemplo anterior por um método chamado **descrever()**, que apresentará todos os atributos de um aluno.

```
public class Aluno {
    String nome;
    int matricula;
    String cpf;
    int idade;

  /* Método descrever, que, com apenas uma chamada, mostrará todos os atributos d
e um objeto */

    public void descrever(){
        System.out.println("Nome do Aluno: "+nome);
        System.out.println("Matrícula do aluno: "+matricula);
        System.out.println("CPF do aluno: "+cpf);
        System.out.println("Idade do aluno: "+idade);
        System.out.println("\n");
    }
}
```

```
public class AlunoApp {

  public static void main(String[] args) {

    Aluno aluno1 = new Aluno();
    Aluno aluno2 = new Aluno();
    Aluno aluno3 = new Aluno();

    aluno1.nome = "Celso Medeiros";
    aluno1.matricula = 1;
    aluno1.cpf = "00134578090";
    aluno1.idade = 23;

    aluno2.nome = "Camila Alves";
    aluno2.matricula = 2;
    aluno2.cpf = "00534578119";
    aluno2.idade = 32;

    aluno3.nome = "Joana Carneiro";
    aluno3.matricula = 3;
    aluno3.cpf = "0019007890";
    aluno3.idade = 17;

    System.out.println("Mostrar Alunos:\n");
/* Chamada da classe descrever() */
    aluno1.descrever();

/* Uma linha de código evita a escrita neste caso de quatro linhas que são:
  System.out.println("Nome do aluno1: " + aluno1.nome);
      System.out.println("Matrícula do aluno1: " + aluno1.matricula);
      System.out.println("CPF do aluno1: " + aluno1.cpf);
      System.out.println("Idade do aluno1: " + aluno1.idade);
  */

    aluno2.descrever();
    aluno3.descrever();
}

}
```

Note que, com bem menos códigos, conseguiu-se o mesmo resultado:


run:
Mostrar Alunos:
Nome do Aluno: Celso Medeiros
Matrícula do aluno: 1
CPF do aluno: 00134578090
Idade do aluno: 23
Nome do Aluno: Camila Alves
Matrícula do aluno: 2
CPF do aluno: 00534578119
Idade do aluno: 32
Nome do Aluno: Joana Carneiro
Matrícula do aluno: 3
CPF do aluno: 0019007890
Idade do aluno: 17

Métodos com passagem de valores e retorno

Outra função extremamente importante dos métodos surge quando é preciso utilizar a passagem de valores por meio dos atributos da classe para calcular algo que será retornado como resposta. Para este exemplo, será modificada a configuração da classe **aluno** para se focar em atributos necessários ao cálculo da média de um aluno. O escopo da classe agora será o seguinte:

ALUNO
NOME
NOTA1
NOTA2
MÉDIA

Tabela 4 – Classe Aluno com argumentos modificados
Fonte: Senac EAD (2022)

Nessa classe, os atributos serão modificados e haverá dois métodos iniciais: um para descrever o “nome”, a “nota1” e a “nota2” do aluno e outro para calcular a média.

```
public class Aluno {
    String nome;
    float nota1;
    float nota2;
    float media;// Aqui, é utilizado o tipo de atributo “float” para aceitar núme
ros com casas decimais

    /* Primeiro método ainda será void apenas para mostrar os dados principais do
aluno */
    public void descrever(){
        System.out.println("Nome do Aluno: "+nome);
        System.out.println("Nota 1: "+nota1);
        System.out.println("Nota 2: "+nota2);
        System.out.println("\n");
    }

    /* método retornando um valor do tipo float, por meio do comando do recebimen
to de dois parâmetros */
    public float calcularMedia(float num1, float num2){
        return (num1 + num2)/2;
    }

}
```

Observe pelo método **calcularMedia()** que estão sendo recebidos dois parâmetros, ambos do tipo *float*. A quantidade de parâmetros para um método é livre e segue o padrão (tipo1 parametro1, tipo2 parametro2,... tipoN parametroN).

Note ainda a presença da palavra **return** ao fim do método. Este é o comando que encerra a execução do método e devolve um valor ao código que o invocou.

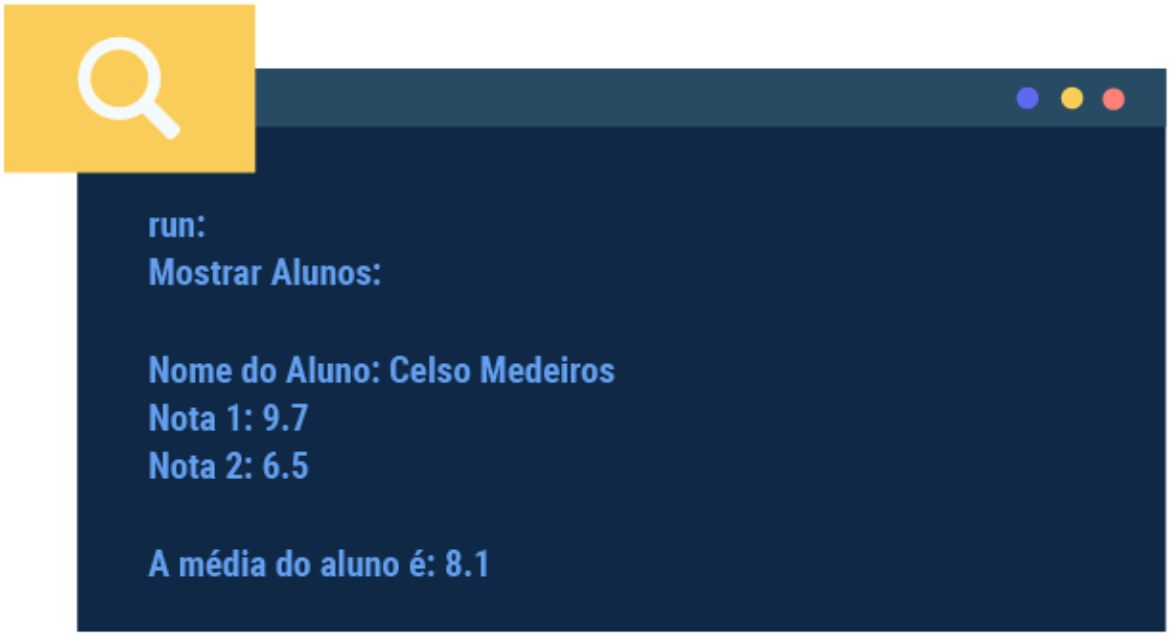
O valor será necessariamente do tipo indicado na assinatura do método – neste caso *float*.

```
public float calcularMedia(float num1, float num2)
```

Quando não se quer retornar nenhum valor, utiliza-se *void* no lugar do tipo na assinatura do método; havendo qualquer tipo diferente de *void*, o método precisará ter um **return** com um valor do tipo indicado.

Agora, note em **main()** da classe principal o uso do método **calcularMedia()**.

```
public class AlunoApp {

    public static void main(String[] args) {

        Aluno aluno1 = new Aluno();

        aluno1.nome = "Celso Medeiros";
        aluno1.nota1=(float) 9.7;
        aluno1.nota2=(float) 6.5;

        System.out.println("Mostrar Alunos:\n");
        aluno1.descrever();
        System.out.println("a média do aluno é: " + aluno1.calcularMedia(aluno1.n
ota1, aluno1.nota2));
    }
}
```



A grande vantagem em se utilizar métodos é que, analisando o exemplo mostrado, é possível instanciar uma turma de, por exemplo, 40 alunos com suas notas e apenas com os métodos **descrever()** e **calcularMedia()**. É possível detalhar as informações

de cada aluno, sem ter que reescrever todos os dados e cálculos de média para cada aluno individualmente.

Outra análise importante desse formato de programação é sua escalabilidade. Imaginem um projeto no qual é necessário inserir outras possibilidades, como, por exemplo, um método para mostrar qual é a maior nota do aluno e outro método para mostrar se o aluno foi aprovado ou reprovado, conforme a média (se maior ou igual a sete). Em um paradigma estruturado, seria preciso fazer esse cálculo individualmente para cada aluno, neste caso, serão criados dois métodos que resolverão o problema, independentemente do número de alunos de uma turma.

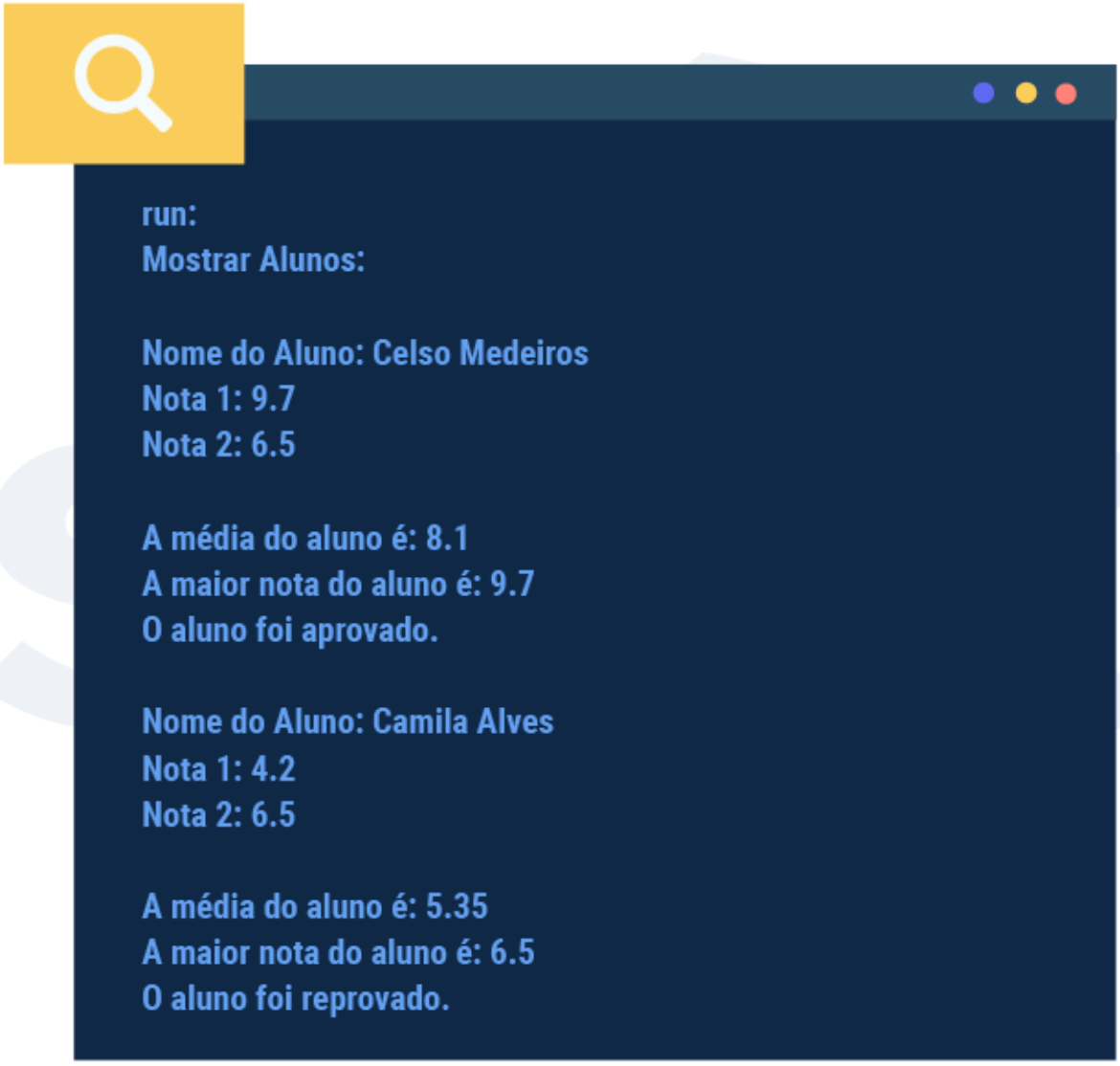
Abaixo do método **calcularMedia()**, da classe **Aluno**, você criará os métodos **mostrarMaior()** e **mostrarResultado()**. O primeiro mostrará a maior nota do aluno e a segundo indicará se ele foi aprovado ou reprovado, conforme suas duas notas apresentadas.

Será utilizado o mesmo código já criado e incluído o que for necessário para resolver este problema.

```
public float mostrarMaior() {
    if (nota1 >= nota2) {
        return nota1;
    } else {
        return nota2;
    }

}

public void mostrarResultado() {
    if (media >= 7) {
        System.out.println("O aluno foi aprovado.\n");
    } else {
        System.out.println("O aluno foi reprovado.\n");
    }
}
```

```
public class AlunoApp {

    public static void main(String[] args) {

        Aluno aluno1 = new Aluno();
        Aluno aluno2 = new Aluno();

        aluno1.nome = "Celso Medeiros";
        aluno1.nota1=(float) 9.7;
        aluno1.nota2=(float) 6.5;

        aluno2.nome = "Camila Alves";
        aluno2.nota1=(float) 4.2;
        aluno2.nota2=(float) 6.5;

        System.out.println("Mostrar Alunos:\n");

        aluno1.descrever();
        System.out.println("A média do aluno é: "+aluno1.calcularMedia());

        System.out.println("A maior nota do aluno é: "+aluno1.mostrarMaior());
        aluno1.mostrarResultado();

        aluno2.descrever();
        System.out.println("A média do aluno é: "+aluno2.calcularMedia());

        System.out.println("A maior nota do aluno é: "+aluno2.mostrar_maior());
        aluno2.mostrarResultado();
    }

}
```

Os métodos têm um grande poder sobre o objeto, podendo modificar o valor de seus atributos e, em outros casos, apenas retornando uma mensagem e não modificando os valores dos atributos dos objetos. A classe **calcularMedia()**, por exemplo, neste caso, retorna o valor do cálculo da média do aluno, mas ao mesmo tempo também modifica o valor do atributo “media da classe” para o resultado encontrado, ou seja, mesmo se o comando **return** não existisse neste método, ele ainda teria um grande poder de afetar este objeto. Este é o resultado na tela:



Note que foi mostrado o resultado para dois alunos, para exemplificar situações em que os alunos foram aprovados ou reprovados. Se você quisesse determinar agora essa situação para 40 alunos, bastava instanciar os 40 objetos com seus valores de atributos e chamar os métodos **descrever()**, **calcularMedia()**, **mostrarMaior()** e **mostrarResultado()**.

Para instanciar 40 objetos, você teria que utilizar os seguintes comandos:

```
Aluno aluno1 = new Aluno();
Aluno aluno2 = new Aluno();
Aluno aluno3 = new Aluno();
Aluno aluno4 = new Aluno();

/* E assim sucessivamente até:
.
.
.
Aluno aluno40 = new Aluno(); */
```

Veja em outro exemplo um programa Java que contém uma classe que simula as operações de um televisor. Considere que a televisão pode ligar, desligar, trocar de canal e alterar o volume do som. Os canais podem ir de 2 a 13 (os canais VHF) e o volume do som vai de 0 a 100. Essas operações só podem ser realizadas com o aparelho ligado.

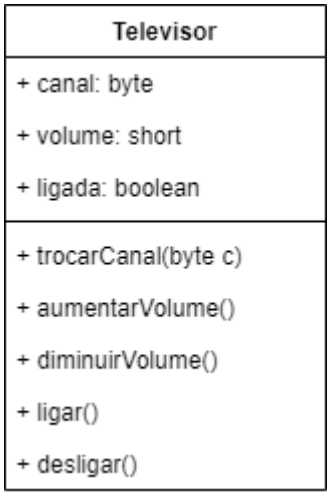


Figura 7 – Diagrama UML para a classe Televisor
Fonte: Senac EAD (2022)

Após criar novo projeto no NetBeans chamado de “TesteTV”, crie a classe **Televisor** sob o pacote “testetv” (ou o pacote principal de seu projeto). Inicie incluindo os atributos.

```
public class Televisor {
    byte canal;
    short volume;
    boolean ligada = false;
}
```

Note que, como variáveis, os atributos podem se inicializados diretamente com um valor padrão (ligada = false).

Implemente agora os métodos ligar e desligar.

```
public class Televisor {
    byte canal;
    short volume;
    boolean ligada = false;

    public void ligar(){
        ligada = true;
        System.out.println("A TV está ligada.");
    }

    public void desligar(){
        ligada = false;
        System.out.println("A TV foi desligada.");
    }
}
```

Os métodos apenas alterarão o atributo “boolean”, que será necessário para as próximas tarefas. Inclua agora, **após o método desligar()** na classe, os métodos para troca de canal e alteração de volume.

```
public class Televisor {
    byte canal;
    short volume;
    boolean ligada = false;

    public void ligar(){
        ligada = true;
        System.out.println("A TV está ligada.");
    }

    public void desligar(){
        ligada = false;
        System.out.println("A TV foi desligada.");
    }

    public void trocarCanal(byte novoCanal) {
        if(!ligada){ //! é o operador "não"
            System.out.println("A TV está desligada!");
        }
        else { //só realizará a operação se a TV estiver ligada
            if(novoCanal > 1 && novoCanal < 14) { //canais VHF vão de 2 a 13
                canal = novoCanal;
                System.out.println("Canal " + canal + " sintonizado");
            }else{
                System.out.println("Canal inválido");
            }
        }
    }

    public void aumentarVolume(){
        if(!ligada){ //só realizará a operação se a TV estiver ligada
            System.out.println("A TV está desligada!");
        }
        else {
            if(volume < 100){ //volume máximo = 100
                volume++;
            }
            System.out.println("Volume atual: " + volume);
        }
    }

    public void diminuirVolume(){
        if(!ligada){ //só realizará a operação se a TV estiver ligada
            System.out.println("A TV está desligada!");
        }
        else {
            if(volume > 0){ //volume mínimo = 0
                volume--;
```

```
            }
            System.out.println("Volume atual: " + volume);
        }
    }
}
```

Por fim, tem-se a interação com o usuário programando o método **main()** da classe principal **TesteTV** para utilizar um objeto “Televisor”.

```
import java.util.Scanner;

public class TesteTV {

    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        String operacao = "";
        byte canal;
        short volume;

        Televisor minhaTV = new Televisor();//instanciando o objeto Televisor
        do
        {
            System.out.println("Digite a operação da TV: [L]igar; [D]esligar; [T]
rocar canal; [+]Aumentar volume; [-]Diminuir volume; [X]Sair");
            operacao = entrada.nextLine();

            switch(operacao)
            {
                case "L":
                    minhaTV.ligar();
                    break;
                case "D":
                    minhaTV.desligar();
                    break;
                case "T":
                    System.out.print("Digite o canal desejado (entre 2 e 13): ");
                    canal = entrada.nextByte();
                    entrada.nextLine(); //consumir o [enter] digitado após o núme
ro
                    minhaTV.trocarCanal(canal);
                    break;
                case "+":
                    minhaTV.aumentarVolume();
                    break;
                case "-":
                    minhaTV.diminuirVolume();
                    break;
                case "X":
                    System.out.println("Encerrando programa");
                    break;
                default:
                    System.out.println("Opção inválida");
                    break;
            }
        }while(!operacao.equals("X"));
    }
```

```
}
```

Teste cada uma das operações para verificar se tudo está funcionando corretamente.

Desafio 1

Altere a classe **Televisor** do exemplo mostrado e inclua um método **display()**, que mostra na tela o canal e o volume atuais se o televisor estiver ligado.

Desafio 2

Crie uma classe chamada **calculadora**, que tenha os atributos “valor1”, “valor2” e “operação”, e um método **calcular()**, que deve retornar o resultado conforme os valores e a operação escolhidos. As operações serão “+” para adição, “-” para subtração, “*” para multiplicação e “/” para divisão.

No último exemplo, você viu uma situação em que foi preciso criar 40 objetos do tipo “Aluno”, e a forma escolhida para isso foi a de criá-los um por um. Porém, dentro da programação, há uma possibilidade que torna essa solução menos trabalhosa e mais organizada. Para tanto, será utilizado neste momento um termo já aprendido no curso, que é o vetor, ou seja, um tipo de variável que possibilita armazenar várias informações do mesmo tipo ao mesmo tempo, mudando apenas o seu endereço na memória.

Veja o exemplo na tabela a seguir:

```
String aluno = new String[5];
```

0	1	2	3	4
Pedro	Ana	Camila	Josué	Jonas

Aqui você deve ter relembrado o conceito de vetor, pois na tabela há cinco nomes de alunos armazenados dentro de uma mesma variável do tipo *string*, mudando apenas seu endereço de memória, como, por exemplo:

```
 aluno[0] = "Pedro";
aluno[1] = "Ana";
aluno[2] = "Camila";
aluno[3] = "Josué";
aluno[4] = "Jonas";
```

Esse conceito pode ser utilizado dentro da POO agora, para auxiliar na criação de vários objetos, os quais são chamados de **vetores de objetos**. É muito importante esse conhecimento quando você precisa instanciar vários objetos ao mesmo tempo em um sistema.

No código a seguir será exemplificado seu uso.

```
public class AlunoApp {

    public static void main(String[] args) {
        /* na próxima linha, cria-se um vetor de objeto com 40 posições do tipo A
luno, ou seja, que utiliza a classe como guia e pode ter todos os seus atributos
utilizados, tais como nome, nota1, nota2 e média, e também a possibilidade de uti
lização de todos os seus métodos  */
        Aluno[] aluno = new Aluno[40];
        int i;

        /* o índice do vetor vai de 0 a 39, o que permite a inserção de 40 objeto
s, desde o objeto aluno[0] até o aluno[39] */
        for(i = 0; i<=39; i++){
        aluno[i] = new Aluno();
    }

    /* para mostrar que essa forma de instância de vários objetos funciona, armaz
ene nome em três objetos diferentes */
    aluno[0].nome="Maria";
    aluno[].descrever();

    aluno[20].nome="José";
    aluno[20].descrever();

    aluno[39].nome="Josi";
    aluno[39].descrever();

}
```

Diferenças entre variáveis de tipo primitivo e de referência

Perceba que classes são tipos e objetos são como variáveis desses tipos. Existem, porém, algumas diferenças importantes entre variáveis e objetos. A principal é interna. As variáveis primitivas (de tipo *int*, *float* etc.) estão representando diretamente uma área na memória, na qual o valor fica armazenado. Já os objetos são o que se pode chamar de “variáveis de referência”; essa variável em si conterá apenas um valor simples, que é o endereço para outra parte da memória – ou seja, uma referência a outra porção de memória, na qual o objeto, de fato, estará alocado.

Ao atribuir uma variável primitiva à outra, você está de fato copiando o valor de uma à outra. A partir daí, se uma variável for alterada, a outra não será afetada.

Figura 8 – Variáveis var1 e var2 do tipo *int*. Atribuir var2 a var1 corresponde a copiar o valor de uma na outra
Fonte: Senac EAD (2022)

Em variável de referência, no entanto, ao atribuir uma variável à outra, você estará apenas fazendo com que **a primeira aponte para onde apontava a segunda**. Ou seja, elas apontarão para um mesmo objeto. Com isso, se, a partir de uma dessas variáveis, o objeto for alterado, consultando a partir da outra variável será possível ver o objeto com a mesma alteração.

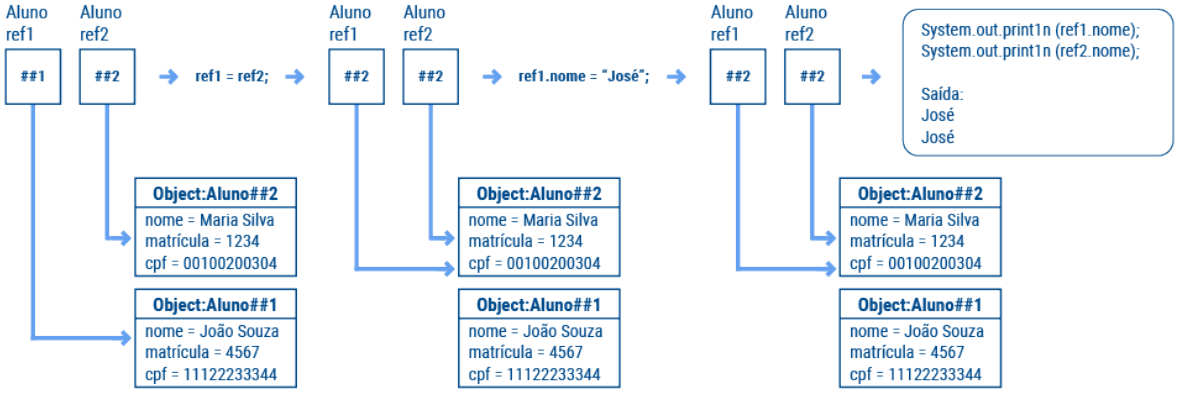


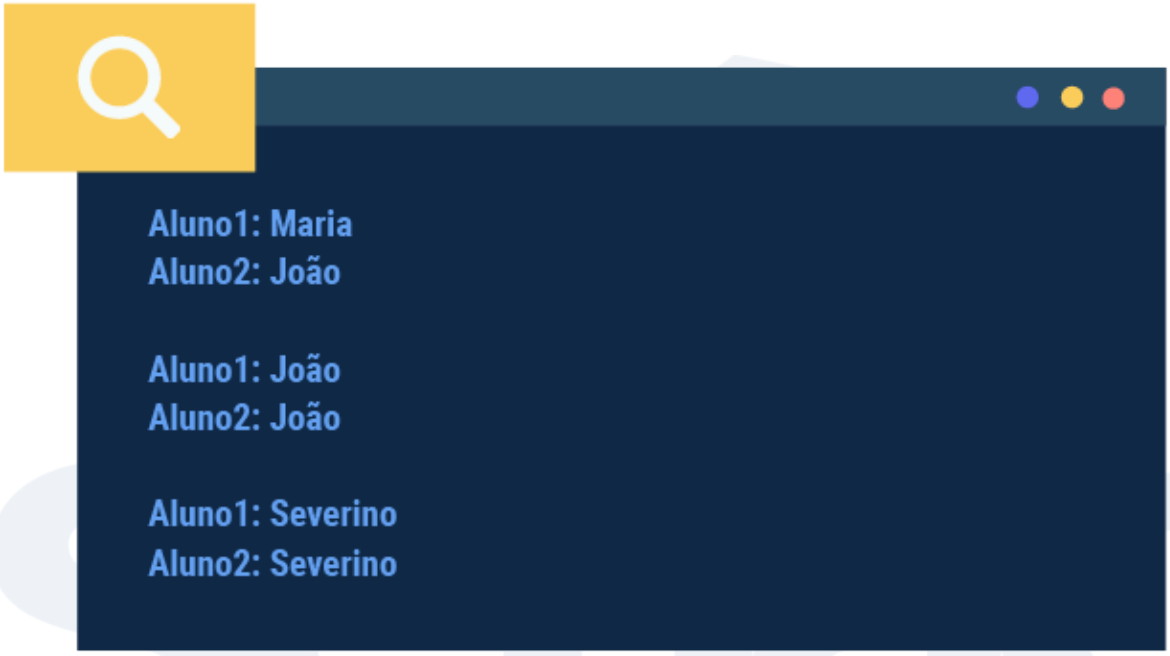
Figura 9 – Variáveis de referência (objetos) ref1 e ref2
Fonte: Senac EAD (2022)

As variáveis iniciam cada uma apontando para seu objeto, mas, ao atribuir a segunda à primeira, o resultado é que ambas acabam apontando (e depois alterando) um mesmo objeto. Ao contrário dos tipos primitivos, a atribuição, neste caso, não é uma cópia do valor, mas sim um apontamento.

Você pode testar essa característica utilizando o seguinte código no método **main()** de seu projeto “AlunoApp”.

```
public static void main(String[] args) {
    Aluno aluno1 = new Aluno();
    Aluno aluno2 = new Aluno();

    aluno1.nome = "Maria";
    aluno2.nome = "João";

    System.out.println("Aluno1: " + aluno1.nome + "\nAluno2: " + aluno2.nome+
"\n");

    aluno1 = aluno2;

    System.out.println("Aluno1: " + aluno1.nome + "\nAluno2: " + aluno2.nome +
"\n");

    aluno1.nome = "Severino";

    System.out.println("Aluno1: " + aluno1.nome + "\nAluno2: " + aluno2.nome+
"\n");
}
```

Ao executar, a saída será algo como o descrito:



Uma variável de referência ainda pode conter valor **null**, o que, na verdade, é a ausência de valor (a variável não aponta para nenhum objeto).

```
Aluno a = null;
```

No exemplo apresentado, se você tentar utilizar a variável, por exemplo, “a.nome = "José"”, obterá um erro do tipo “NullPointerException”, já que não há um objeto “Aluno” a ser manipulado, apenas um vazio.

Classe, atributos e métodos, objetos

Assista agora um vídeo que explica na teoria e na prática o uso de classes, o que são atributos e como instanciar um objeto.


CH presencial:
Recurso

Métodos construtores

São métodos chamados sempre que um novo objeto é criado e possibilita uma série de comandos. os quais permitem a inicialização dos valores que serão guardados nos atributos. É importante frisar que esse tipo de método **não retorna nenhum tipo de valor**.

Após a definição dos atributos de uma classe, é preciso criar no código o seu método construtor, para possibilitar a criação dos objetos que são a instância de uma classe. Pode-se transformar a abstração em algo real e que ajude o usuário na resolução de um problema.

Uma mesma classe pode ter vários métodos construtores, usando ou não todos os atributos dela, pois essa propriedade permite que sejam criados diversos tipos de objetos por meio de uma mesma estrutura abstrata de uma classe. Com os códigos mostrados em seguida, você terá uma melhor visão de como funciona um método construtor.

Relembre a classe **Aluno**, criada anteriormente para entender como funciona a criação de um método construtor. Nessa classe, constam os seguintes atributos:

- “Nome” – *String*
- “Matrícula” – *Int*
- “Curso” – *String*

Sempre que for preciso criar novos objetos no programa, deve-se utilizar os métodos construtores desta classe por meio de linhas de código, como as mostradas a seguir.

```
public class Aluno {
    String nome;
    int matricula;
    String curso;

    /* o método construtor sempre ficará abaixo dos atributos da classe */
    public Aluno(String nome,int matricula, String curso){
        this.nome=nome;
        this.matricula=matricula;
        this.curso=curso;
    }
}
```

O método construtor **sempre terá o mesmo nome da classe** à qual ele se refere.

No código criado anteriormente, visualiza-se um método construtor que recebe três parâmetros (nome, matrícula e curso), e o modo de inicializar esses valores ocorre por meio da expressão **this**, que é uma forma de o Java se referir ao valor que está vindo do objeto e será recebido pelo método.

Veja na prática como isso funciona. Para compreender, é necessário primeiro aprender como se cria um objeto para chamar esse método construtor, conforme as linhas de código a seguir:

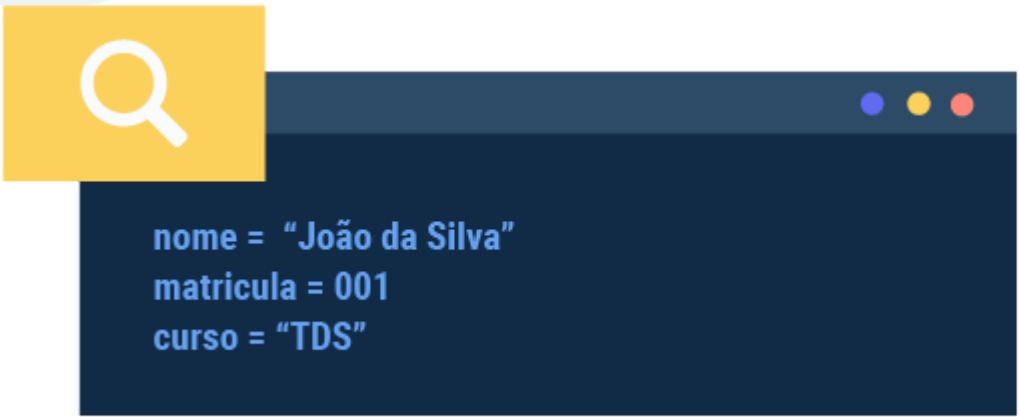
```
public class AlunoApp {
    /*aqui, há outra classe, na qual serão criados os objetos */

    public static void main(String[] args) {
        /* o método main é o método principal, necessário para que sua aplicação
Java funcione; se ele não for declarado, já aparecerá um erro e seus objetos não
serão criados */

        Aluno aluno1 = new Aluno("João da Silva",001,"TDS");
        Aluno aluno2 = new Aluno("Jorge Aguiar",002,"TDS");
        Aluno aluno3 = new Aluno("Camila Santos",003,"TDS");
    }
}
```

O símbolo “//” serve para comentar uma linha de código em Java e o símbolo “/*” no início e */” no final servem para comentar uma ou mais linhas de código ao mesmo tempo.

Analise a seguinte linha de código:

```
Aluno aluno1 = new Aluno("João da Silva",001,"TDS");
```

Aqui, foi criado um objeto chamado “aluno1”, que é do tipo **Aluno** (classe criada anteriormente), e utilizada a expressão reservada **new** para a criação desse objeto “aluno” com os seguintes parâmetros:



A partir desse momento, essas informações serão passadas para o método construtor, que armazenará os valores na memória deste objeto, como no seguinte exemplo:

```
/* o método receberá os valores “João da Silva”, 001 e “TDS” */
public Aluno(String nome,int matricula, String curso){

    this.nome=nome;
    /* this.nome é o valor recebido em String nome e possui armazenado “João da S
ilva”, e este valor será armazenado no atributo nome do objeto aluno1 */

    this.matricula=matricula;
    /* this.matricula é o valor recebido em int matricula e possui armazenado 00
1, e este valor será armazenado no atributo matricula do objeto aluno1 */

    this.curso=curso;
    /* this.curso é o valor recebido em String curso e possui armazenado “TDS”, e
este valor será armazenado no atributo curso do objeto aluno1 */

}
```

Considerando os códigos mostrados, há três objetos instanciados sem a necessidade de iniciar nome, matrícula e curso separadamente para cada aluno.

```
Aluno aluno1 = new Aluno("João da Silva",001,"TDS");
Aluno aluno2 = new Aluno("Jorge Aguiar",002,"TDS");
Aluno aluno3 = new Aluno("Camila Santos",003,"TDS");
```

Nem sempre é preciso preencher todos os atributos de um objeto quando ele é iniciado, por isso, em uma mesma classe, é possível ter vários métodos construtores com o mesmo nome da classe, porém recebendo apenas os atributos necessários. Por exemplo, uma classe pode ter atributos nome, matrícula e curso e dois métodos construtores, em que um recebe todos os três atributos e outro recebe apenas nome e matrícula, sem a inserção do curso.

Agora, observe como é criado mais de um construtor para a mesma classe e como isso pode ser utilizado em um programa. Para tanto, será utilizada uma classe chamada “Carro”, com quatro atributos.

```
public class Carro {
    String modelo;
    String marca;
    int ano;
    String cor;
}
```

Com base na classe carro, será demonstrada a utilização de mais de um tipo de método construtor:

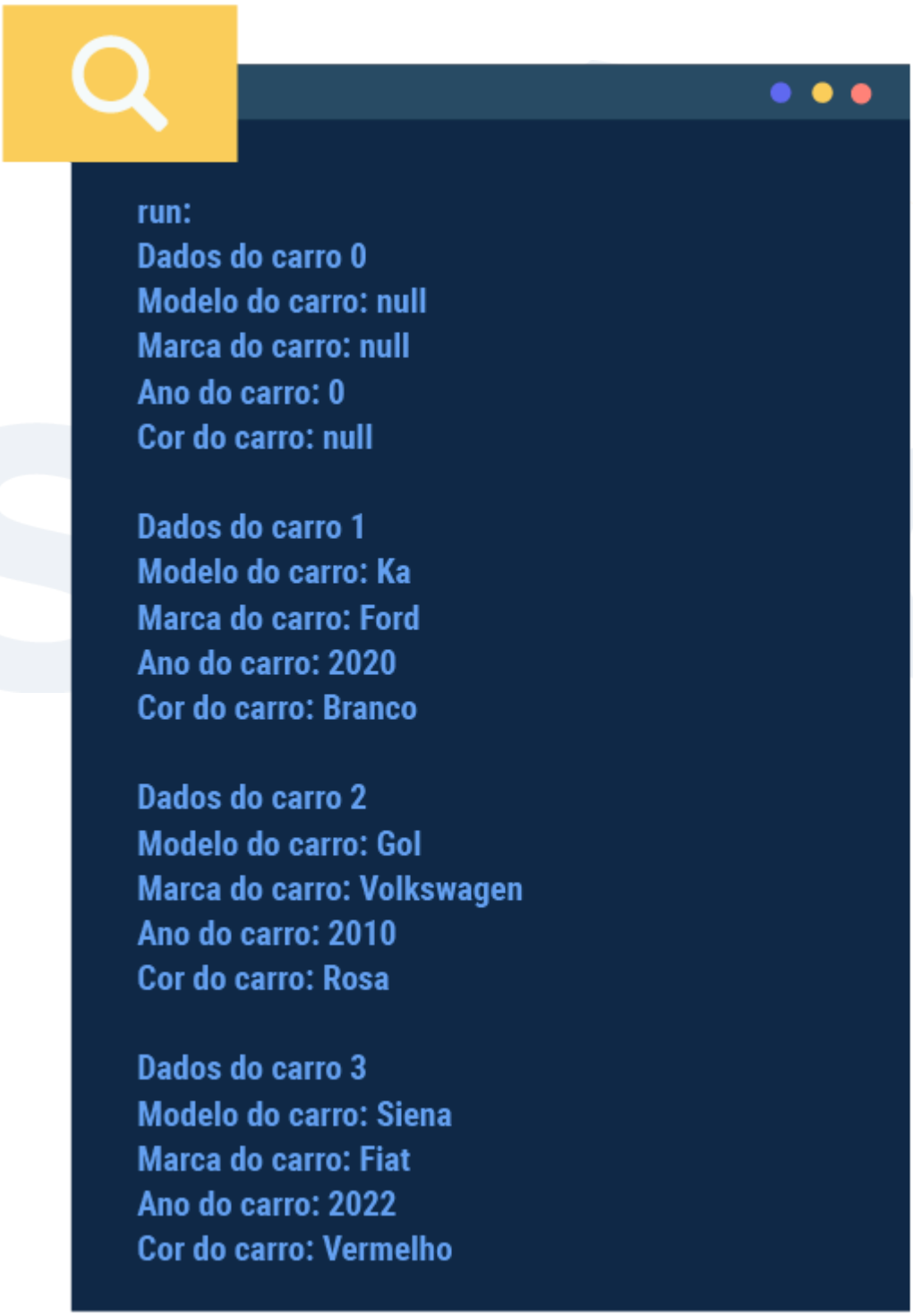
```
public class Carro {
    String modelo;
    String marca;
    int ano;
    String cor;

    public Carro(){
    /* Este é um método padrão, chamado quando um objeto é criado, sem passagem d
e valores de atributos */
    }

    /* Construtor com quatro parâmetros, que recebe todos os atributos sempre que
instancia um novo objeto*/
    public Carro(String modelo, String marca, int ano, String cor){
        this.modelo=modelo;
        this.marca=marca;
        this.ano=ano;
        this.cor=cor;
    }

    /* Construtor com três parâmetros, que recebe três atributos e define a cor i
nicial automaticamente para vermelha*/
    public Carro(String modelo, String marca, int ano){
        this.modelo=modelo;
        this.marca=marca;
        this.ano=ano;
        this.cor="Vermelho";
    }


    public void descrever(){
        System.out.println("Modelo do carro: "+modelo);
        System.out.println("Marca do carro: "+marca);
        System.out.println("Ano do carro: "+ano);
        System.out.println("Cor do carro: "+cor);
        System.out.println("\n");
    }

}
```

Para o exemplo não ficar muito extenso, foram criados três construtores, mas poderia ter outros, como, por exemplo, com um ou dois atributos, conforme a necessidade de utilização do sistema.

Para visualizar melhor a importância dessa variedade de construtores, observe o código a seguir, no qual serão criados os objetos utilizando os diversos tipos de construtores:

```
public static void main(String[] args) {
    Carro carros[] = new Carro[4];
    carros[0] = new Carro();
    carros[1] = new Carro("Ka", "Ford", 2020, "Branco");
    carros[2] = new Carro("Gol", "Volkswagen", 2010, "Rosa");
    carros[3] = new Carro("Siena", "Fiat", 2022);

    for(int i = 0; i < carros.length; i++)
    {
        System.out.println("Dados do carro " + i);
        carros[i].descrever();
    }

}
```

O resultado apresentado na tela será:



Analisando o retorno do programa, há importantes considerações a se fazer, como o fato de o “carro[0]” ter iniciado modelo, marca e cor como “**null**” e ano como “**0**”. Isso ocorreu por se ter utilizado o primeiro método construtor, que guarda o espaço de memória para o objeto recém-criado, mas não tem nenhuma passagem de valor, deixando os atributos vazios, com a possibilidade de modificação de

valores em outro momento. Outra informação importante é o fato de o “carro[3]” não ter utilizado cor na passagem de parâmetro e sua cor ter sido instanciada como **vermelho**. Isso ocorreu porque se utilizou **o construtor de três parâmetros**, que inicializa a cor em “vermelho” automaticamente.

Os outros três carros utilizaram o construtor de quatro parâmetros e tiveram a passagem completa de valores escolhidos pelo usuário do sistema.

Métodos Construtores

Assista agora um vídeo que explica na teoria e na prática o que são Métodos Construtores.

Associações entre objetos

Um sistema não será formado apenas por uma classe e provavelmente várias dessas classes se relacionarão com outras. Chamam-se “associação” os arranjos em que uma classe contém alguma ligação (geralmente um atributo) com outra classe.

Imagine que é preciso representar funcionários de uma empresa com o seguinte código (crie um novo projeto “TesteEmpresa” para acompanhar):

```
import java.time.LocalDate;

public class Funcionario {
    String nome;
    String cargo;
    LocalDate nascimento;
    double salario;

    public Funcionario(String nome, String cargo, LocalDate nascimento, double sa
lario) {
        this.nome = nome;
        this.cargo = cargo;
        this.nascimento = nascimento;
        this.salario = salario;
    }
}
```

Observe que está sendo utilizada uma classe da biblioteca de Java – **LocalDate** – para armazenar data. Por ser externa, ela necessita do **import** para ser localizada (**import java.time.LocalDate;**).

No projeto, precisa-se ainda representar uma empresa, que tem nome e CNPJ. A empresa também tem um gerente, que é um funcionário. É possível representar essa situação em Java do seguinte modo:

```
public class Empresa {
    String nome;
    String cnpj;
    Funcionario gerente;
}
```

Com isso, estabelece-se uma associação entre “Empresa” e “Funcionario”. Você poderia ir além e representar na classe a lista de funcionários que a empresa emprega:

```
public class Empresa {
    String nome;
    String cnpj;
    Funcionario gerente;
    Funcionario[] funcionarios;

    public Empresa(String nome, String cnpj, int numeroFuncionarios)
    {
        funcionarios = new Funcionario[numeroFuncionarios];
        this.nome = nome;
        this.cnpj = cnpj;
    }
}
```

Desta vez, o atributo é um vetor de objetos do tipo “Funcionario”. Foi criado ainda um construtor que recebe o nome, o CNPJ e a quantidade de funcionários empregados, valor a partir do qual já foi instanciado um vetor com a quantidade exata. Veja a manipulação dessas classes no código de **main()** do projeto.

```
public class TesteEmpresa {

    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        Empresa novaEmpresa;

        novaEmpresa = new Empresa("XYZ S.A.", "13.522.667/0001-07", 3);

        Funcionario joao = new Funcionario("João", "Gerente", LocalDate.of(1980,
10, 10), 2000.5);

        novaEmpresa.gerente = joao;

        //o vetor "funcionarios" foi construído com três posições no construtor
        novaEmpresa.funcionarios[0] = joao;
        novaEmpresa.funcionarios[1] = new Funcionario("Maria", "Vendas", LocalDat
e.of(1990, 10, 15), 1500);
        novaEmpresa.funcionarios[2] = new Funcionario("Joaquim", "Vendedor", Loca
lDate.of(1997, 1, 30), 1500);

        //acessando recursos dos objetos associados a Empresa
        System.out.println("A empresa " + novaEmpresa.nome
                + " conta com " + novaEmpresa.gerente.nome + " na gerência"
                + " e possui " + novaEmpresa.funcionarios.length + " funcionario
s: ");

        //percorrendo o vetor “funcionarios" e acessando informações de cada obje
to guardado nele
        for(int i = 0; i&lt; novaEmpresa.funcionarios.length; i++)
        {
            System.out.println("\t" + novaEmpresa.funcionarios[i].nome + "(" + no
vaEmpresa.funcionarios[i].cargo + ")");
        }

    }

}
```

Repare, pelo código, que está sendo atribuído um objeto “joao”, do tipo “Funcionario”, ao atributo “gerente” de “novaEmpresa”, do tipo “Empresa”. A partir daí, é possível acessar o objeto por **novaEmpresa.gerente** e os atributos ou

métodos disponibilizados por esse objeto (**novaEmpresa.gerente.nome**, por exemplo). Pelo exemplo, note como se dá a manipulação de um atributo que é uma lista de objetos (no caso **novaEmpresa.funcionarios**).

LocalDate.of() é um método da classe **LocalDate**, que cria um objeto de data de acordo com os parâmetros, nesta ordem: ano, mês, dia. Assim, **LocalDate.of(1990, 10, 15)**, por exemplo, representa a data 15/10/1990.

No mesmo projeto mostrado, crie uma nova classe **Venda**, que mantenha o total vendido (*double*) e o vendedor que a realizou.

Pacotes Java (*packages*)

Uma maneira de organizar o código do projeto é separá-lo em **pacotes**, áreas virtuais que agrupam classes. Em Java, um pacote simboliza um caminho de pastas no qual está o código-fonte da classe.

Por padrão, ao criar um projeto no NetBeans, gera-se um pacote de mesmo nome do projeto.

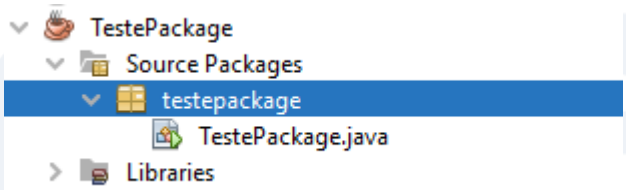


Figura 10 – Pacote “testepackage” criado para o projeto TestePackage no Netbeans
Fonte: NetBeans 13 (c2017-2020)

É possível criar novos pacotes por meio do NetBeans clicando com o botão direito do *mouse* sobre o projeto e selecionando **New** > **Java Package**. Dentro do novo pacote, pode-se incluir uma nova classe clicando com o botão direito do *mouse* sobre o pacote e selecionando **New** > **Java Class**.

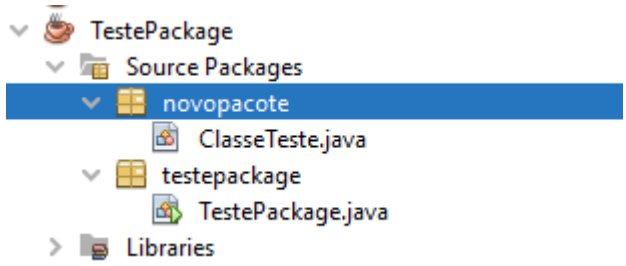


Figura 11 – Pacote “novopacote” criado e, nele, uma nova classe “ClasseTeste”
Fonte: NetBeans 13 (c2017-2020)

Um pacote pode receber qualquer nome, mas há um certo padrão em nomear, como uma URL (*uniform resource locator*) ao contrário, por exemplo, “br.com.senac.utilitarios” – e, neste caso, será criado um caminho de pastas br/com/senac/utilitários, em que ficarão armazenadas as classes do pacote.

A classe pertencente a um pacote precisa ter essa informação indicada no topo do arquivo, como o exemplo da “ClasseTeste” da figura anterior:

```
package novopacote;

public class ClasseTeste {
    public int numero;
}
```

Quando não há uma linha *package* no arquivo, o Java considera que a classe está em um “pacote padrão”.

As classes pertencentes a um mesmo pacote se conhecem e podem se referenciar. No entanto, se estiverem em pacotes diferentes, é necessário usar o comando **import**. Para exercitar, crie um novo pacote chamado “br.com.teste” e, nele, uma classe chamada **ClasseTeste2**.

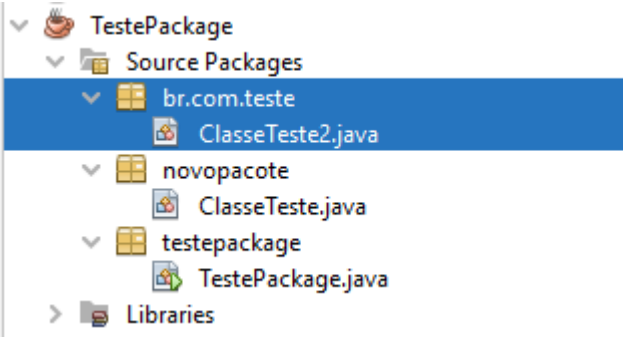


Figura 12 – Novo pacote “br.com.teste” contendo novo arquivo ClasseTeste2.java
Fonte: NetBeans 13 (c2017-2020)

Na classe **ClasseTeste** de “novopacote”, será incluído um novo atributo do tipo “ClasseTeste2”.

```
public ClasseTeste2 teste;
```

Ao incluir, você poderá notar erro no NetBeans com sugestões de correção, uma delas “Add import for br.com.teste.ClasseTeste2”. Essa sugestão pode ser aceita, o que resulta o seguinte:

```
package novopacote;

import br.com.teste.ClasseTeste2;


public class ClasseTeste {

    public int numero;

    public ClasseTeste2 teste;

}
```

O **import**, portanto, é necessário para que uma classe tenha acesso à outra que está em um pacote diferente. É possível importar as classes do pacote uma a uma ou todas de uma vez, como a seguir:

```
import br.com.teste.*;
```

No entanto, essa não é uma prática recomendável, já que o programa perderá tempo com acesso a outras classes que não interessam ao código. O recomendável é um **import** para cada classe necessária.

O uso de pacotes, portanto, é bastante recomendado para projetos maiores, para melhor separação de classes com propósitos relacionados e para casos nos quais é necessário incluir classes de mesmo nome no projeto (não pode haver duas classes com mesmo nome em um mesmo *package*, mas em diferentes sim).

Modificadores de acesso

É de extrema importância que qualquer programador do paradigma de orientação a objetos tenha pleno entendimento deste conceito, pois os modificadores de acesso permitem definir quais classes e membros (métodos e variáveis da classe) terão visibilidade diante de outras. Neste caso, quando se fala em visibilidade, faz-se, na verdade, referência à possibilidade de acesso – somente o que está visível pode ser acessado.

Dentro do Java há quatro tipos de **modificadores de acesso**, que são ***public***, ***private***, **padrão** e **protected**. Com o uso desses modificadores, é possível definir tudo o que estará ou não visível dentro do sistema. Portanto, você deve conhecer o que significa cada uma dessas propriedades para aplicá-las em seus códigos.

Public

Esse modificador de acesso é o menos restritivo de todos, pois ele permite que qualquer outra parte de sua aplicação tenha acesso ao membro definido como *public*.

Private

Esse modificador de acesso é o mais restritivo de todos, já que ele só permite o acesso dentro da própria classe na qual é declarado.

Padrão ou *default*

Esse modificador permite o acesso dentro do pacote no qual ele está inserido, ou seja, todas as classes ou interfaces de classes dentro do pacote no qual foi criado. Chama-se padrão os membros da classe na qual não foi atribuído outro modificador como *public*, *private* ou *protected*. Esse é o modificador aplicado nos exemplos das classes implementadas neste conteúdo até o momento.

Protected

Esse modificador permite o acesso dentro do pacote no qual ele está inserido e, diferentemente do modo padrão, também permite o acesso por classes derivadas, mesmo que estejam em outros pacotes.

Nas duas imagens a seguir, tem-se uma visualização completa das possibilidades de acesso:

Figura 13 – Tabela de visibilidade dos modificadores de acesso
Fonte: Araújo (2021)

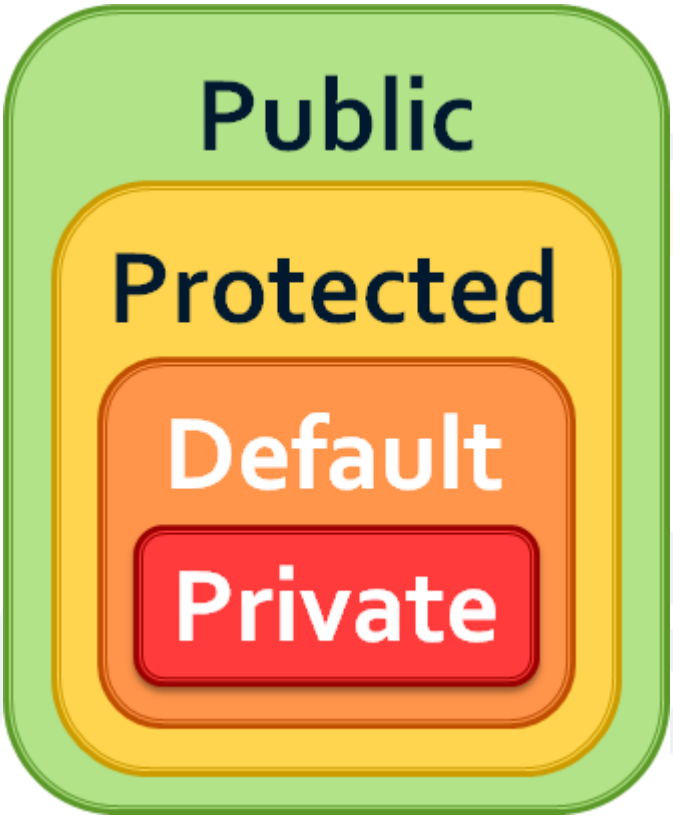


Figura 14 – Ordem de restrição dos modificadores Java
Fonte: Mauda (2017)

A figura 14 mostra a ordem de restrição dos modificadores de acesso, indo do menos restritivo que é o *public*, até o mais restritivo, que é o *private*. A utilização correta deles é um elemento fundamental na segurança do projeto, pois a definição das propriedades de visibilidade e acesso permitem um controle de **disponibilidade, integridade, confidencialidade e autenticidade dos dados**.

A melhor forma de entender esses conceitos é visualizando as linhas de código, onde ele é utilizado. Então, que tal começar a programar? Será preciso retornar ao projeto de alunos.

```
public class Aluno {

    String nome;
    float nota1;
    float nota2;
    float media;
}
```

Ao analisar o código mostrado, note que a classe **Aluno** é do tipo *public*, ou seja, pode ser acessada por qualquer parte do sistema. Já os atributos são do tipo padrão (*default*), que são acessados somente pelas classes do pacote no qual ele está inserido. Sempre que não for informado um modificador de acesso, considera-se “acesso padrão”.

O que acontece se você modificar um atributo para o tipo *private*?

```
public class Aluno {
    private String nome;
    float nota1;
    float nota2;
    float media;
}
```

```
public class AlunoApp {

    public static void main(String[] args) {
        Aluno aluno1 = new Aluno();

        aluno1.nome = "Celso Medeiros";
        aluno1.nota1=(float) 9.7;
        aluno1.nota2=(float) 6.5;

        System.out.println("Mostrar Aluno:\n");
        System.out.println("O nome do aluno é: "+aluno1.nome);
        System.out.println("A nota do aluno é: "+aluno1.nota1);
        System.out.println("A nota do aluno é: "+aluno1.nota2);
    }

}
```

Quando se tenta rodar esse código, ele apresenta um erro, como o mostrado a seguir:

```
Exception in thread "main" java.lang.RuntimeException: Uncompilable code - nome has
private access in alunoapp.Aluno
```

Significa que foi impossível acessar o atributo “nome”, pois ele está definido como *private*, e só seria possível acessá-lo dentro da classe no qual ele foi criado, que é o arquivo “Aluno.java” (onde está a classe **Aluno**), e você estaria tentando acessá-lo do arquivo “AlunoApp.java”.

Neste momento, você pode se perguntar qual é o objetivo em se restringir uma classe e não deixar ela ser acessada. A resposta é simples: por motivo de segurança, existem informações importantes em um programa, as quais não devem ser acessadas ou modificadas pelo usuário sem que antes elas passem por algum tipo de controle. Para acessar esses métodos ou atributos do tipo *private*, será utilizado um conceito de extrema importância, que é o encapsulamento.

Retome o projeto “TesteTV”, exemplificado anteriormente. Proteja seus atributos para que modificações em valores de “canal”, “volume” e “ligada” aconteçam apenas a partir de seus métodos públicos.

Encapsulamento

É um conceito que permite esconder detalhes de implementação de uma classe, deixando acessível ao usuário somente operações consideradas seguras e que mantenham o objeto em um estado correto de funcionamento, ou seja, nunca se deve permitir uma operação que afete negativamente os objetos, gerando erro nas ações do projeto.

Pense em um televisor, por exemplo. Você tem acesso aos controles de volume, canais, ligar e desligar etc., mas não tem acesso a como tudo isto é feito na parte interna do aparelho.

Os objetos podem proteger seu estado (suas informações), declarando atributos com acesso *private*. Para acessar esses atributos, utilizam-se métodos conhecidos como **GET** e **SET**, que, quando chamados, fazem a alteração dentro da própria classe, o que é permitido no modificador de acesso *private*.

Um método **GET** (pegar) é utilizado quando se acessa algum atributo, e deve sempre ter retorno do mesmo tipo do atributo a ser manipulado. Já o método **SET** (alterar) é utilizado para modificar o valor de algum atributo e sempre conta com um parâmetro do mesmo tipo do atributo manipulado. Ambos os métodos devem ser públicos, para que as outras classes os acessem ao invés de alcançar diretamente atributos. Esses métodos garantem maior controle da classe, permitindo, por exemplo, validações sobre os valores informados.

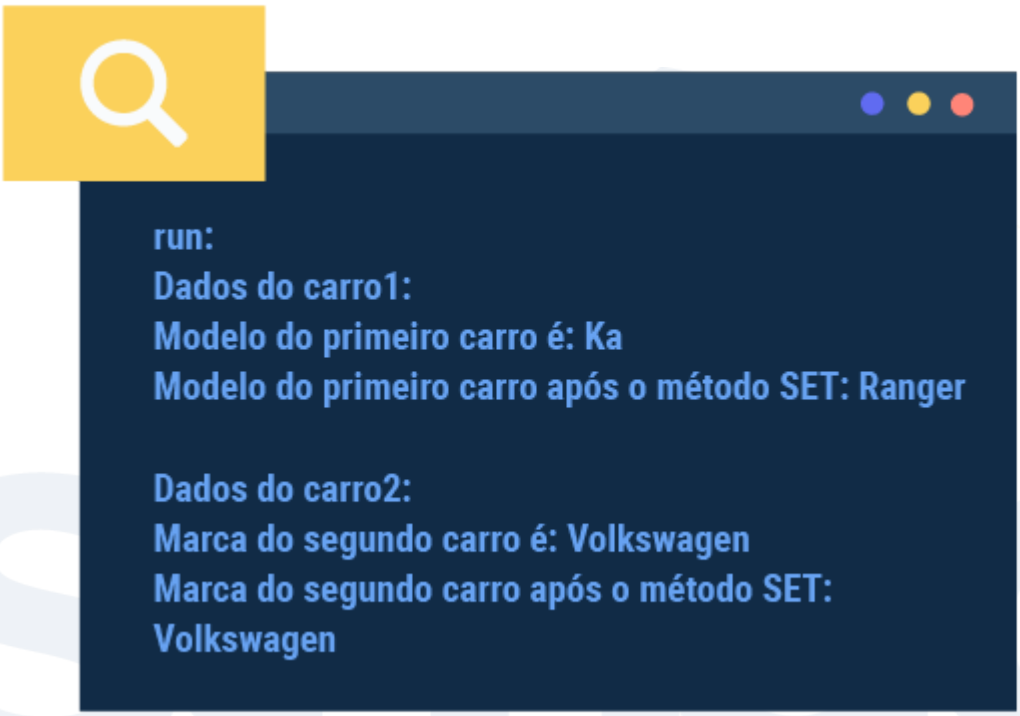
Volte à classe **Carro**, criada anteriormente, mas agora utilizando os conceitos aprendidos em Atributos, Métodos, Métodos construtores, Modificadores de acesso e Encapsulamento.

```
/* aqui já começam as modificações, todos os atributos definidos como private, au
mentando a segurança e não permitindo que o usuário inicie ou altere seu valor, a
menos que seja por meio de um método GET ou SET criado na própria classe Carro */

public class Carro {
    private String modelo;
    private String marca;
    private int ano;
    private String cor;
    /* Neste caso, será usado apenas um método construtor com quatro atributos, p
ara deixar o código mais limpo e focar em todos os conceitos */

    public Carro(String modelo, String marca, int ano, String cor){
        this.modelo=modelo;
        this.marca=marca;
        this.ano=ano;
        this.cor=cor;
    }


    /* Neste momento, será criado um método GET e SET para cada um dos atributos
da classe Carro */

    /* Aqui, o primeiro método GET (getModelo) criado para o atributo modelo, do
tipo String e tem uma única função, que é retornar o valor do atributo modelo*/
    public String getModelo() {
        return modelo;

    }

    /* Aqui, o primeiro método SET (setModelo) criado para o atributo modelo, que
recebe uma String como parâmetro e inicializa ou atualiza o valor do atributo mod
elo e a linha this.modelo faz referência à própria classe, igual ao método modifi
cador */
    public void setModelo(String modelo) {
        this.modelo = modelo;

    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public int getAno() {
```

```
        return ano;
    }

    public void setAno(int ano) {
        this.ano = ano;
    }

    public String getCor() {
        return cor;
    }

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

```
public class CarroApp {

    public static void main(String[] args) {

        Carro carro1 = new Carro("Ka", "Ford", 2020, "Branco");
        Carro carro2 = new Carro("Gol", "Volkswagen", 2010, "Rosa");


        System.out.println("Dados do carro1: ");
        System.out.println("Modelo do primeiro carro é: "+carro1.getModelo());
        carro1.setModelo("Ranger");
        System.out.println("Modelo do primeiro carro após o método SET: "+carro1.
getModelo());


        System.out.println("\nDados do carro2: ");
        System.out.println("Marca do segundo carro é: "+carro2.getMarca());
        carro2.setModelo("Volks");
        System.out.println("Marca do segundo carro após o método SET: "+carro2.ge
tMarca());

    }

}
```

O resultado na tela após rodar o programa será o seguinte:



Analise detalhadamente as seguintes linhas:

```
Carro carro1 = new Carro("Ka", "Ford", 2020, "Branco");
Carro carro2 = new Carro("Gol", "Volkswagen", 2010, "Rosa");
```

Nessas linhas, dois objetos são instanciados utilizando o método construtor da classe **Carro**, que permite preenchimento de suas informações iniciais. Após os objetos estarem iniciados com seus valores dos atributos “modelo”, “marca”, “ano” e “cor”, utilizam-se os métodos **GET** e **SET** para manipular ou visualizar seus valores.

```
System.out.println("Dados do carro1: ");
System.out.println("Modelo do primeiro carro é: "+carro1.getModelo());
carro1.setModelo("Ranger");
System.out.println("Modelo do primeiro carro após o método SET: "+carro1.getModel
o());
```

Os seguintes métodos estão sendo utilizados:

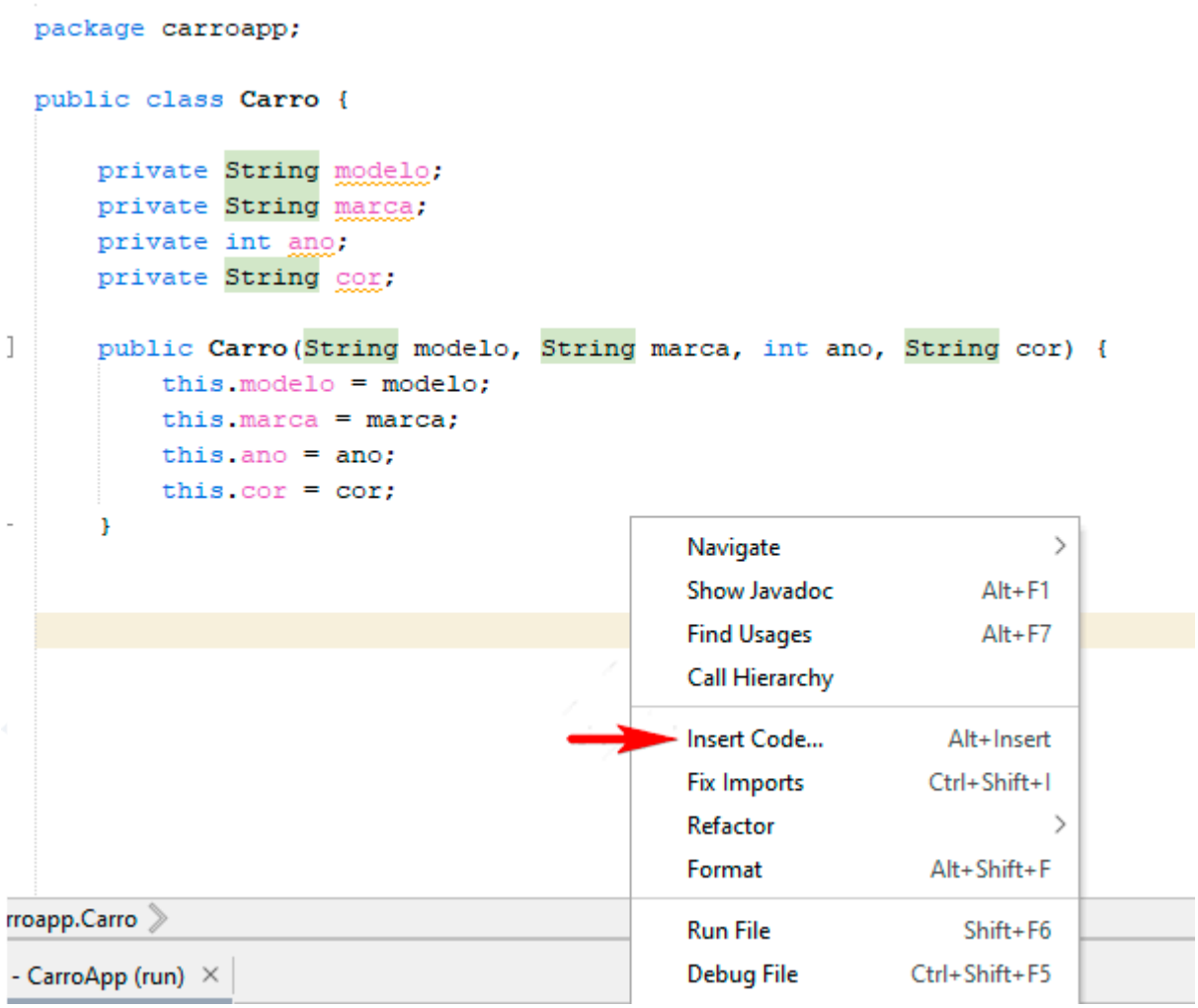
- **carro1.getModelo()**: retornará o valor do atributo “modelo” de “carro1”, que neste programa é **Ka**.
- **carro1.setModelo("Ranger")**: modifica o modelo do “carro1” de **Ka** para **Ranger**.
- **carro1.getModelo()**: serve para retornar o valor agora modificado pelo método **SET**, e que virou o valor **Ranger**.

Criando métodos GET e SET facilmente no NetBeans

É importante, nas primeiras vezes, escrever linha por linha os métodos **GET** e **SET** para ganhar afinidade com as particularidades da criação dos métodos. Mas, como uma boa IDE de desenvolvimento, o NetBeans tem uma forma automática de criação dos métodos **GET** e **SET**, que você verá a seguir:

Passo 1

Após o método construtor na classe **Carro**, ou qualquer outra classe que está sendo criada, clique com o botão direito do *mouse* e escolha o item **insert code**, ou utilize o atalho **Alt + Insert**.



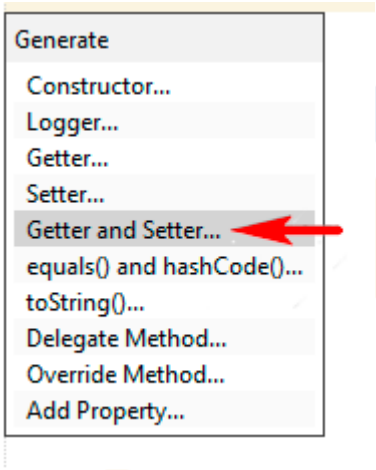
(objetos/fig15.png)

Figura 15 – Menu para inserir códigos automáticos no NetBeans

Fonte: NetBeans 13 (c2017-2020)

Passo 2

Depois de clicar no item **Insert Code**, uma nova janela abrirá, na qual você deve clicar no item **Getter ans Setter...**.

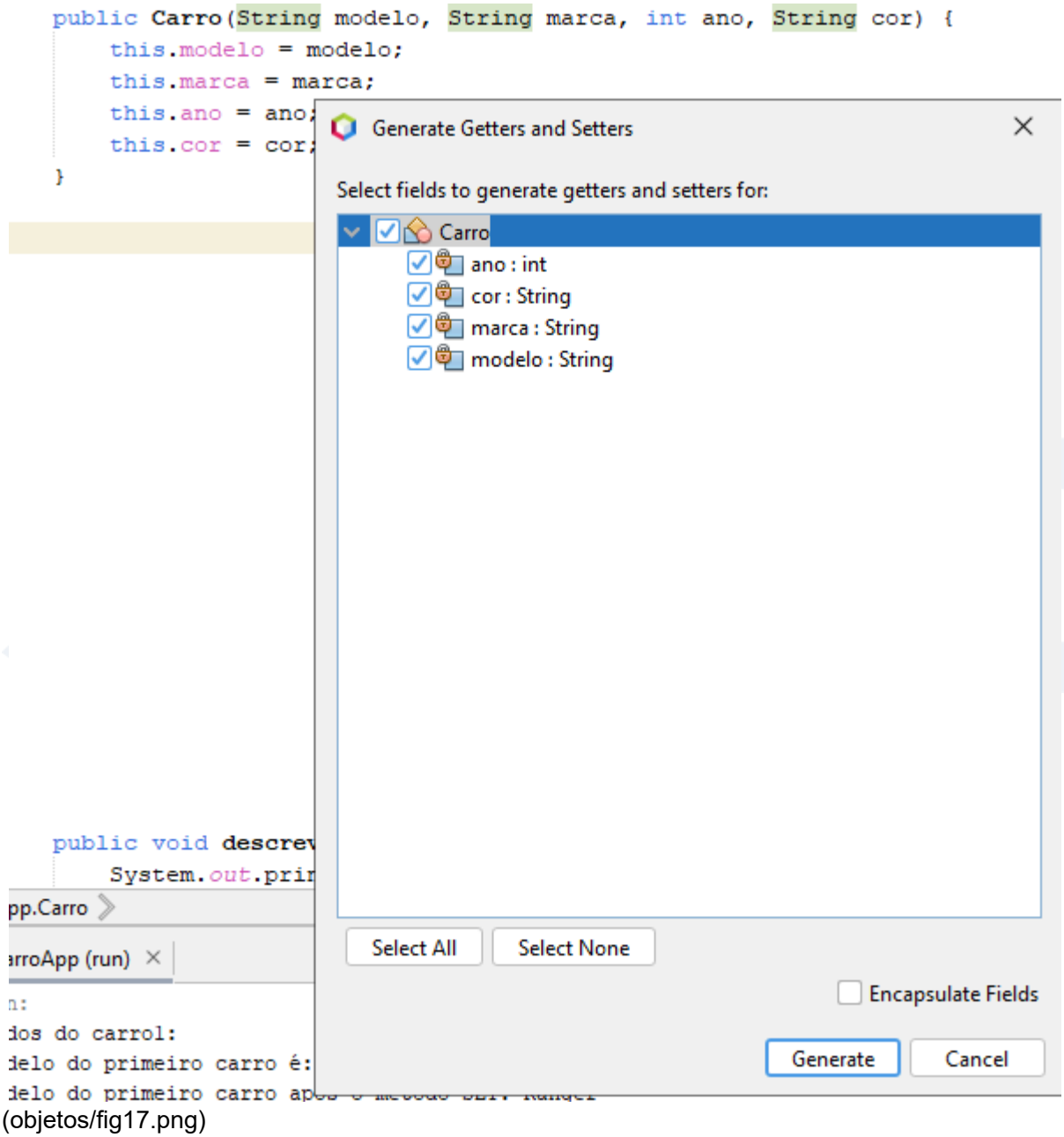


(objetos/fig16.png)

Figura 16 – Menu para inserir os métodos **GET** e **SET** no Netbeans
Fonte: NetBeans 13 (c2017-2020)

Passo 3

Aparecerá uma última janela, em que será preciso escolher a quais atributos serão relacionados os métodos **GET** e **SET**, e, caso você marque o nome da classe, serão criados automaticamente os métodos para todos os atributos da classe.



(objetos/fig17.png)

Figura 17 – Janela de escolha dos atributos para gerar os métodos **GET** e **SET**

Fonte: NetBeans 13 (c2017-2020)

Retome o projeto “TesteEmpresa”, proposto no exemplo da seção sobre a associação entre objetos, e proteja os atributos da classe **Empresa** e da classe **Funcionario**, tornando-os privados, mas acessíveis a partir dos métodos **get()** e **set()**. Realize as alterações necessárias em **main()** para essa nova abordagem.

Modificadores e Encapsulamento

Assista agora um vídeo que explica na teoria e na prática o que são Modificadores de acesso e Encapsulamento.

Antes de encerrar este conteúdo, treine seus conhecimentos avançando nas etapas do *quiz* a seguir, o qual guia a montagem de uma classe Java.

Encerramento

Agora que você aprendeu a lógica de programação orientada a objetos, já está preparado para criar códigos mais complexos e que permitem melhor reutilização, segurança e escalabilidade. O domínio deste conceito é imprescindível a todo programador e, certamente, a atualização deste conhecimento é algo que acompanhará você ao longo de toda sua carreira no mercado de trabalho.