Passo 1 (Entendendo conceitos de orientação a objetos) Desenvolver softwares utilizando o paradigma conhecido como **orientação a objetos** envolve:

1.1 Observação de partes de interesse do mundo real, o que dá origem a o que alguns autores chamam de **minimundo**. Aquilo que faz parte de seu minimundo depende do problema que está resolvendo. Por exemplo, um sistema acadêmico poderia envolver alunos, professores, disciplinas etc. Um sistema usado por um banco poderia envolver clientes, contas, produtos financeiros como empréstimos, investimentos etc.

1.2 Representação das partes de interesse utilizando componentes de software. Uma vez detectadas as partes de interesse do mundo real, cabe ao programador descrevê-las de modo que possam ser utilizadas posteriormente. Aqui, o programador obtém modelos ou descrições do que lhe é de interesse. Na orientação a objetos, esses modelos são conhecidos como **classes**.

1.3 Uma vez feita a modelagem, ou seja, obtenção das classes de interesse, cabe ao programador construir **objetos** a partir delas, os quais, por sua vez, interagem entre si a fim de fazer com que o sistema funcione.

Passo 2 (Examinando alguns exemplos de classes e objetos)

2.1 Suponha que você deseja **dirigir um carro**. Antes de mais nada, o carro precisa ser projetado, o que envolve desenhos de engenharia, detalhes sobre suas partes elétricas, mecânicas etc. Essa descrição ou modelo (são sinônimos) é uma **classe** a partir da qual carros podem ser construídos. Terminada a descrição do veículo, cada um deles construído a partir dela é um **objeto** do tipo carro.

2.2 Suponha que uma construtora deseja construir uma **nova casa**. Antes de mais nada, é preciso elaborar um documento que descreva suas medidas, seus detalhes de implementação. Ou seja, a planta da casa. A planta da casa, ou seja, seu modelo ou descrição, é uma **classe** a partir da qual **objetos** do tipo casa podem ser construídos.

2.3 O ser humano possui características próprias. Em um sistema de software, é perfeitamente possível descrever partes de interesse que digam o que é um ser humano. Ou seja, é possível escrever um modelo que descreva um ser humano: tem dois olhos, duas pernas, uma cabeça que pode ter cabelo etc. Essa descrição é uma **classe** a partir da qual **objetos** do tipo ser humano podem ser construídos.

Note que uma classe é uma mera descrição. Nesses exemplos, uma casa propriamente dita (concreta) ou um carro propriamente dito (concreto) são objetos construídos a partir de suas classes.

Passo 3 (Entendendo atributos e métodos) O que compõe uma classe?

3.1 Classes descrevem características (**atributos**) de seus objetos.

3.1.1 Um **carro** possui cor, ano de fabricação, modelo, fabricante, número de portas, informações sobre seus acessórios etc. Esses são alguns possíveis **atributos**.

3.1.2 Uma **casa** possui cor, número de cômodos, medida de cada cômodo, número de andares, valor venal, tipo de telhado etc. Esses são alguns possíveis **atributos**.

3.1.3 Uma **pessoa** tem cor de olhos, cor de pele, altura, peso, número que calça, sexo, idade etc. Esses são alguns possíveis **atributos**.

3.2 Classes também descrevem os comportamentos (**métodos**) de seus objetos.

3.2.1 Um **carro** é algo inanimado, porém, outros objetos podem interagir com ele, fazendo com que ele acelere, vire à esquerda, dê seta, estacione etc. Esses são alguns comportamentos possíveis de um carro.

3.2.2 Uma **casa** também é inanimada. Porém, outros objetos podem interagir com ela fazendo com que seja fechada uma de suas portas, seja aberta uma janela etc. Também podemos fazer uma casa falar ou dormir, caso ela pertença a um jogo (um tanto criativo). Tudo depende do contexto (o minimundo) de interesse.

3.2.3 Um **ser humano** é capaz de falar, dormir, estudar, namorar, piscar os olhos, escrever entre muitas outras coisas. Cada uma dessas atividades é um de seus possíveis comportamentos.

Nota 3.1: Cada comportamento citado de cada classe é implementado por meio do uso de **métodos.**

Nota 3.2: Quando desejamos instruir um objeto a realizar determinada tarefa, **enviamos uma mensagem a ele** ou **chamamos um método sobre ele**. Essas duas expressões são sinônimas.

Passo 4 (Implementando um livro de notas) Um professor deseja fazer o controle de notas de seus alunos usando um software personalizado. Iremos escrever esse software para ele, de maneira incremental. A cada passo iremos adicionar novas funcionalidades.

4.1 Comece criando um novo projeto.

4.2 Crie uma nova classe cujo nome deverá ser **LivroDeNotas**. Essa classe descreve as características desejáveis em um livro de notas. Veja sua implementação inicial na Listagem 4.2.1.

Listagem 4.2.1

```
public class LivroDeNotas {

}
```

4.3 Um **comportamento** de nosso livro de notas será exibir uma mensagem de boas vindas ao professor que o utilizar. Para implementar um comportamento, usamos um método, lembra? Dessa forma, escreva o método **exibirMensagem** como mostra a Listagem 4.3.1. Note que o nome do método deve ser escolhido de modo a promover a legibilidade do código e que ele deve respeitar o padrão *camel case*.

Listagem 4.3.1

```
public class LivroDeNotas {
   public void exibirMensagem (){
      System.out.println ("Bem-vindo ao livro de notas");
   }
}
```

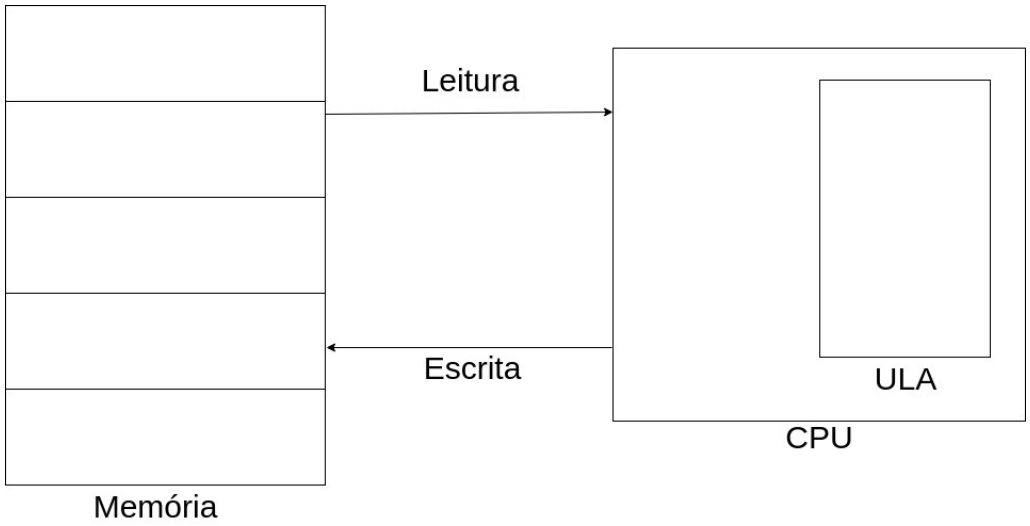
4.4 Agora iremos criar uma classe para **testar** a classe livro de notas. Ela terá o método main e nele iremos criar um objeto do tipo LivroDeNotas e, sobre ele, chamaremos o método exibirMensagem. A razão pela qual optamos por criar uma nova classe para o teste (ao invés de criar o método main na classe LivroDeNotas mesmo) envolve um princípio conhecido como **alta coesão**. Quando escrevemos uma classe, desejamos que ela tenha um único propósito, uma única razão de ser, que resolva um único problema. A classe LivroDeNotas já possui um propósito: ela representa livros de notas. Adicionar o método main a ela implicaria em dar-lhe uma nova responsabilidade: servir para o teste. Violar esse princípio, em geral, implica em redução dos níveis de **reusabilidade, flexibidade, manutenabilidade** entre outros de nossas classes. É comum que classes possuam relativamente um número pequeno de linhas de código. Crie a classe da Listagem 4.4.1. Veja que ela possui um método **main**. Nele, criamos um objeto do tipo **LivroDeNotas** e enviamos a mensagem **exibirMensagem** para ele, usando uma **variável de referência** que o referencia após ter sido criado pelo operador **new**.

Listagem 4.4.1

```
public class TesteLivroDeNotas {
   public static void main(String[] args) {
      LivroDeNotas livroDeNotas = new LivroDeNotas();
      livro.exibirMensagem();
   }
}
```

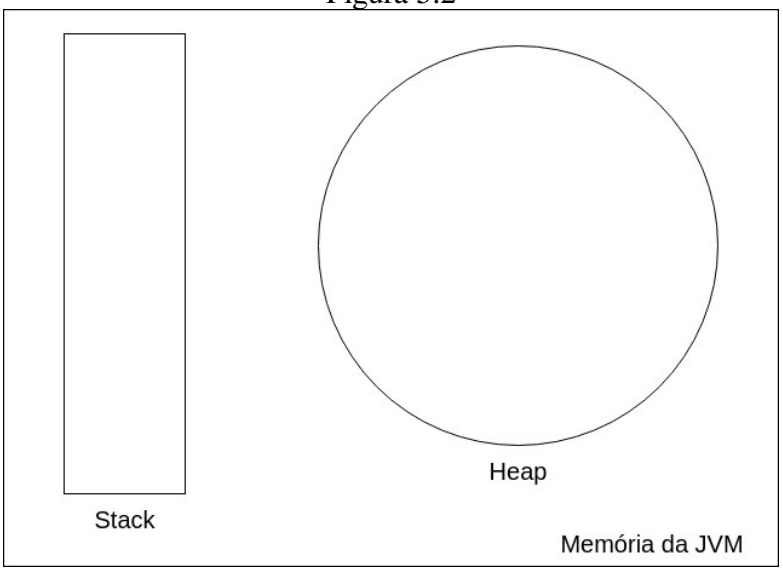
Passo 5 (O modelo de memória da JVM) Lembre-se de que um computador, de maneira simplificada, opera utilizando dois componentes: sua CPU e sua memória principal. A CPU executa instruções aritméticas e lógicas e utiliza a memória para armazenar valores necessários para essas operações, seus resultados etc. Veja a Figura 5.1.

Figura 5.1



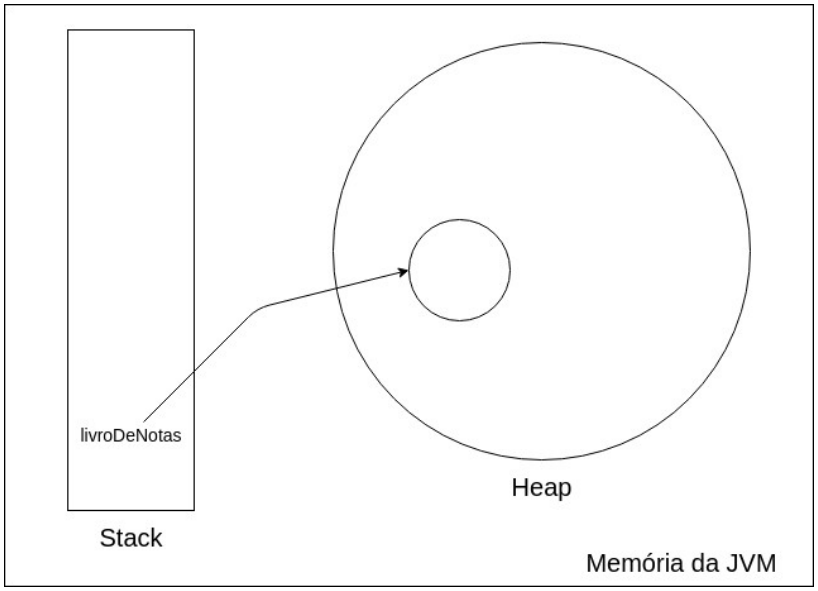
Quando a máquina virtual Java (JVM) entra em cena, o sistema operacional reserva um espaço de memória para que ela possa funcionar. A JVM, por sua vez, organiza a memória da forma exibida pela Figura 5.2, que mostra duas regiões importantes da memória: a região **stack** e a região **heap**. Note que na região stack ficam as variáveis de referência e que, na região heap, ficam os objetos. Uma variável de referência referencia um objeto na memória heap para que ele possa ser manipulado. Objetos para os quais não exista uma variável de referência não podem ser utilizados. A figura geométrica escolhida para cada região não tem importância. O que é importante é entender que cada uma delas representa um "pedaço" de memória reservado para a JVM e que, em geral, a memória heap fica com mais espaço do que a memória stack.

Figura 5.2



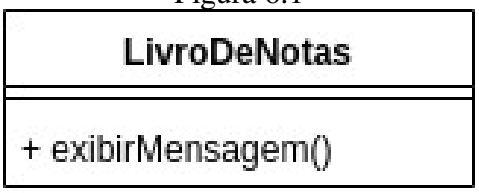
No programa que escrevemos até então, temos uma variável de referência e um objeto, ambos do mesmo tipo (LivroDeNotas). Desta forma, quando a JVM está em execução e o interpreta, a memória fica como exibe a Figura 5.3.

Figura 5.3



Passo 6 (Descrevendo a classe LivroDeNotas graficamente) É muito comum descrever os mais diferentes componentes de um sistema orientado a objetos utilizando uma linguagem gráfica denominada UML (*Unified Modelling Language*). É um meio de expressar partes importantes de um sistema, de maneira independente de sua implementação. Assim, mesmo não programadores podem avaliar e entender os componentes do sistema. A Figura 6.1 mostra um diagrama de classes que mostra, graficamente, a classe LivroDeNotas. Note que há 3 compartimentos na classe. O superior mostra o nome da classe. O intermediário contém os atributos (nossa classe não tem atributos ainda, portanto, esse compartimento está vazio) e o último compartimento mostra os métodos da classe.

Figura 6.1



Passo 7 (Mostrando o nome do curso por meio de um parâmetro de método) Iremos agora alterar a classe LivroDeNotas de modo que, ao exibir as boas vindas, ela mostre também o nome do curso que o professor estiver ministrando. Professores de cursos diferentes poderão fazer uso do livro, então o nome do curso não poderá ficar fixo na classe. Utilizaremos o recurso chamado **parâmetro de método** para deixar aberto esse valor e permitir que a classe cliente (neste caso, a classe TesteLivroDeNotas) especifique o valor a ser exibido. Adapte a classe LivroDeNotas como mostra a Listagem 7.1.

Listagem 7.1

```
public class LivroDeNotas {
   public void exibirMensagem (String nomeDoCurso){
      System.out.printf ("Bem-vindo ao livro de notas do curso %s", nomeDoCurso);
   }
}
```

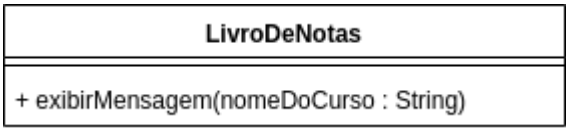
Para testar, será necessário enviar o nome do curso como um **argumento** para o método. Veja a Listagem 7.2.

Listagem 7.2

```
public class TesteLivroDeNotas {
   public static void main(String[] args) {
      LivroDeNotas livroDeNotas = new LivroDeNotas();
      String nomeDoCurso = JOptionPane.showInputDialog ("Prof, qual o nome do
curso?");
      livroDeNotas.exibirMensagem(nomeDoCurso);
   }
}
```

Passo 8 (Diagrama de classes com método com parâmetro) A classe LivroDeNotas agora pode ser representada graficamente (usando a linguagem UML) como mostra a Figura 8.1.

Figura 8.1



Passo 9 (Usando uma variável de instância para o nome do curso) Ao longo da execução do programa, talvez seja de interesse permitir que o nome de curso associado a um livro de notas seja utilizado mais de uma vez ou mesmo alterado. Por essa razão, iremos utilizar um recurso chamado de **variável de instância** (instância é sinônimo de objeto) para especificar que cada objeto do tipo LivroDeNotas possui um atributo que representa o nome do curso. É muito comum (mas não uma regra absoluta) que variáveis de instância sejam marcadas com o modificador de acesso **private**. Ele indica que aquela variável somente pode ser acessada pelos métodos da classe em que foi declarada e serve para a implementação de um recurso chave da orientação a objetos conhecido como **encapsulamento**. Aprenderemos que esse recurso está intimamente relacionado com o princípio conhecido como **baixo acoplamento**, que indica que duas classes que interagem entre si sabem poucos detalhes de implementação uma da outra, o que quer dizer que, se uma tiver sua implementação alterada, a outra pode continuar funcionando sem precisar ser alterada também. Para permitir que as classes clientes manipulem indiretamente suas variáveis de instância, uma classe pode oferecer os conhecidos métodos **getters/setters**. A alteração aparece na Listagem 9.1.

Listagem 9.1

```
public class LivroDeNotas {
   private String nomeDoCurso;

   public void exibirMensagem (){
      System.out.printf ("Bem-vindo ao livro de notas do curso %s",
getNomeDoCurso());
   }

   public String getNomeDoCurso() {
      return nomeDoCurso;
   }

   public void setNomeDoCurso(String nomeDoCurso) {
      this.nomeDoCurso = nomeDoCurso;
   }

}
```

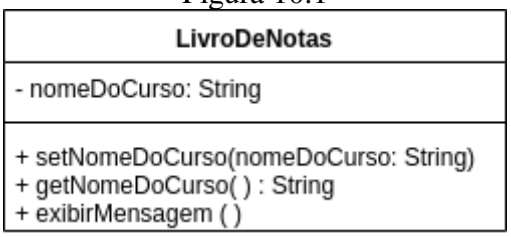
A classe da Listagem 9.2 testa a nova versão do livro de notas.

Listagem 9.2

```
public class TesteLivroDeNotas {
   public static void main(String[] args) {
      LivroDeNotas livroDeNotas = new LivroDeNotas();
      String nomeDoCurso = JOptionPane.showInputDialog ("Prof, qual o nome
do curso?");
      livroDeNotas.setNomeDoCurso(nomeDoCurso);
      livroDeNotas.exibirMensagem();
   }
}
```

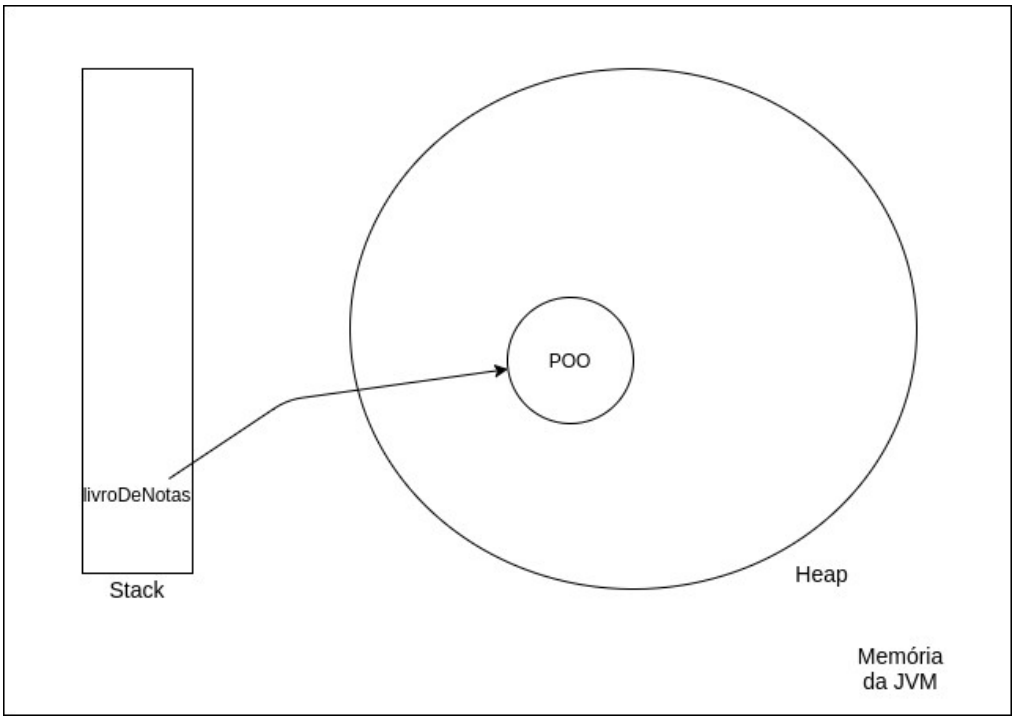
Passo 10 (Diagrama de classes com variável de instância e métodos getters/setters) O diagrama da Figura 10.1 mostra, graficamente, a nova versão da classe LivroDeNotas.

Figura 10.1



Passo 11 (Variável de instância na memória da JVM) Quando tivermos um objeto LivroDeNotas construído e com nome atribuído, o que teremos na memória é aquilo que exibe a Figura 11.1.

Figura 11.1



Passo 12 (Inicializando uma variável de instância com o construtor) Quando utilizamos o operador **new**, como vimos, estamos construindo um objeto da classe envolvida na instrução. Isso é feito por um bloco denominado **construtor**. O que temos feito é utilizar o **construtor padrão** da classe, que o compilador nos entrega gratuitamente. Porém, quando um livro de notas é construído, a princípio ele não tem valor nenhum no campo nomeDoCurso e, posteriormente, temos de fazer a atribuição. Por meio da escrita de um construtor personalizado podemos construir o objeto e fazer a atribuição de valores de variáveis de instância simultaneamente. A classe exibida na Listagem 12.1 faz uso de um construtor assim. O operador **this** permite diferenciar a variável de instância da variável do parâmetro, chamada de **variável local**.

Listagem 12.1

```
public class LivroDeNotas {
   private String nomeDoCurso;

   public LivroDeNotas (String nomeDoCurso){
      this.nomeDoCurso = nomeDoCurso;
   }

   public void exibirMensagem (){
      System.out.printf ("Bem vindo ao livro de notas do curso %s",
getNomeDoCurso());
   }

   public String getNomeDoCurso() {
      return nomeDoCurso;
   }

   public void setNomeDoCurso(String nomeDoCurso) {
      this.nomeDoCurso = nomeDoCurso;
   }
}
```

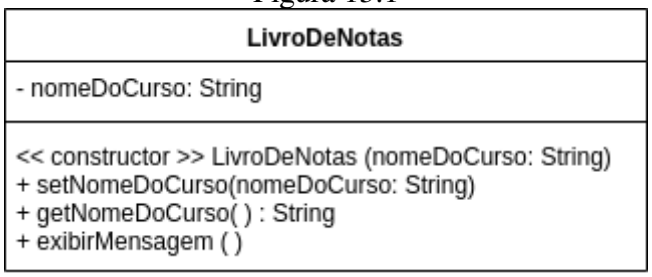
A classe da Listagem 12.2 testa a nova versão da classe TesteLivroDeNotas. Note que estamos construindo dois objetos do tipo LivroDeNotas e que cada um possui seu próprio nomeDeCurso. É por isso que nomeDeCurso é uma variável de instância; cada instância (ou objeto) tem sua própria cópia, independente de qualquer outra instância.

Listagem 12.2

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

      String primeiroCurso = JOptionPane.showInputDialog ("Prof, qual o nome
do primeiro curso?");
      String segundoCurso = JOptionPane.showInputDialog("Prof. qual o nome do
segundo curso?");

      LivroDeNotas livroDeNotas1 = new LivroDeNotas(primeiroCurso);
      LivroDeNotas livroDeNotas2 = new LivroDeNotas (segundoCurso);

      livroDeNotas1.exibirMensagem();
      livroDeNotas2.exibirMensagem();
   }
}
```

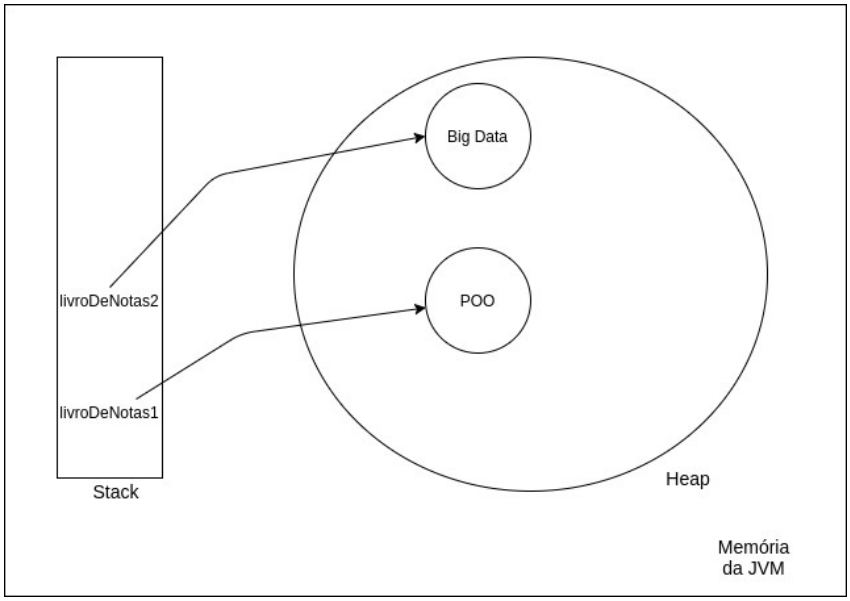
Passo 13 (Diagrama de classes com construtor) A Figura 13.1 mostra, graficamente, a nova classe LivroDeNotas.

Figura 13.1



Passo 14 (Dois objetos na memória) A Figura 14.1 mostra o estado da memória da JVM após dois objetos terem sido construídos.

Figura 14.1



Exercícios

1. Escreva uma classe para representar carros. Adicione a ela dois atributos e dois métodos que lhe pareçam razoáveis. Os dois atributos devem ser encapsulados. Escreva métodos getters/setters para cada um deles.

2. Escreva uma classe de teste que:

2.1 Instancia dois veículos.
2.2 Obtém valores para seus atributos e os atribui adequadamente.
2.3 Chama cada um dos métodos que você criou.
2.4 Exibe os valores das variáveis, usando os métodos getters.

3. Rescreva a classe do exercício 1 adicionando a ela um construtor que recebe valores a serem atribuídos às duas variáveis de instância da classe carro.

4. Note que a classe de teste deixou de funcionar após a adição do construtor. Faça os ajustes necessários para que ela volte a funcionar.

Referências

DEITEL, P. e DEITEL, H. **Java Como Programar**. 8ª Edição. São Paulo, SP: Pearson, 2010.

LOPES, A. e GARCIA, G. **Introdução à Programação – 500 Algoritmos Resolvidos**. 1ª Edição. São Paulo, SP: Elsevier, 2002.