



Trilha de Programação Orientada a Objetos

OT9 – Construção da casa (construir casa)

INDICADORES

- Desenvolver capacidades linguísticas de modo a saber usar adequadamente a linguagem oral e escrita em diferentes situações e contextos;
- Conhecer o caráter do conhecimento científico aplicando a metodologia científica e utilizando redação acadêmica na realização da pesquisa, na escolha de métodos, técnicas e instrumentos de pesquisa;
- Compreender e aplicar técnicas de relações humanas visando o desenvolvimento da liderança e relacionamento em equipe, preservando aspectos éticos e organizacionais;
- Gestão das atividades;
- Cumprimento dos prazos;
- Analisa e avalia o funcionamento de computadores e periféricos em ambientes computacionais;
- Codifica programas computacionais utilizando lógica de programação e respeitando boas práticas de programação;
- Utilizar estruturas de dados definindo-as e aplicando-as adequadamente nos programas;
- Desenvolver sistemas em diferentes segmentos;
- Sistematizar o desenvolvimento de software na concepção do projeto de software.

FICHAMENTO

Classe ArrayList

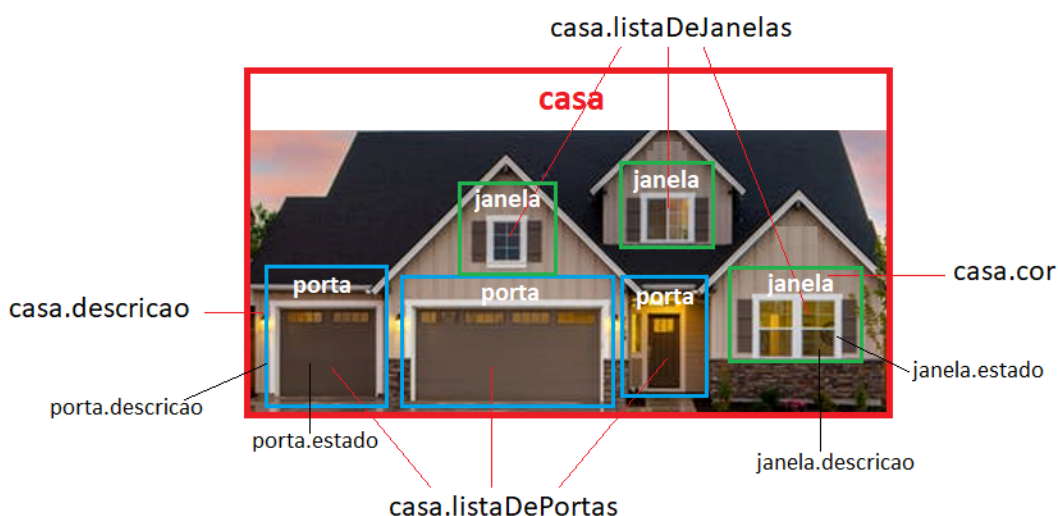
- Definição (utilidade)
- Como um ArrayList insere os dados (ordem)
- Métodos:
 - *add(E e)*
 - *remove(int index)*
 - *size()*
 - *isEmpty()*
 - *contains(Object o)*
 - *get(int index)*
 - *set(int index, E element)*
 - *toArray()*

- Explique com suas palavras a diferença de funcionamento do método `add()` e `set()`

Use a referência da documentação oficial no site da (<https://docs.oracle.com/en/>) para seu fichamento, porém, procure utilizar outra fonte confiável também como 2ª citação, haja vista que na documentação oficial o conteúdo é bastante técnico.

CONSTRUIR CASA

Nessa OT faremos o desenvolvimento da funcionalidade “Construir casa”, que já deixamos disponível em nosso menu, cuja estrutura geral foi criada anteriormente. Para criarmos essa funcionalidade, precisaremos efetivamente, modelar nossas classes que estão no pacote **modelo**, pois seus conteúdos ainda são vazios. Sendo assim, vamos começar pelas classes **Porta** e **Janela**, que correspondem às nossas **Aberturas**. Pois bem, como já discutido na OT anterior, a classe **Aberturas** servirá de referência para as outras duas classes, pois, lembrando nosso desenho, ambas possuem um atributo **descrição** e **estado**:



Sendo assim, na classe **Aberturas**, implemente:

```

package modelo;

public abstract class Aberturas {

    protected String descricao;
    protected int estado;

    public String getDescricao() {
        return descricao;
    }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
    public int getEstado() {
        return estado;
    }
    public void setEstado(int estado) {
        this.estado = estado;
    }
}

```

Agora, nas classes que serão filhas da classe **Aberturas**, é necessário que esse relacionamento seja estabelecido, então, abra as classes **Porta** e **Janela** e faça a devida extensão:

<pre> package modelo; public class Porta extends Aberturas { } </pre>	<pre> package modelo; public class Janela extends Aberturas { } </pre>
--	---

Veja que para fins de economia de espaço e devido às referidas classes não conterem nenhum conteúdo especializado, apenas herdado, podemos fechar a classe na mesma linha de abertura. Relembrando, essa relação entre Aberturas e as classes Porta e Janela evita o seguinte cenário (**não implemente**)...

```
public class Janela {

    private String descricao;
    private int estado;

    public String getDescricao() {
        return descricao;
    }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
    public int getEstado() {
        return estado;
    }
    public void setEstado(int estado) {
        this.estado = estado;
    }
}

public class Porta {

    private String descricao;
    private int estado;

    public String getDescricao() {
        return descricao;
    }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
    public int getEstado() {
        return estado;
    }
    public void setEstado(int estado) {
        this.estado = estado;
    }
}
```

...ou seja, repetição de código em duas classes, o que já vimos que não é interessante, e com o recurso da herança pode ser minimizado. Haja vista que já modelamos nossas aberturas da casa, podemos modelar agora a classe que é o coração de nossa proposta: a classe **Casa**. Implemente:

```

package modelo;
import java.util.ArrayList;

public class Casa {

    private String descricao;
    private String cor;
    private ArrayList<Aberturas> listaDePortas = new ArrayList<Aberturas>();
    private ArrayList<Aberturas> listaDeJanelas = new ArrayList<Aberturas>();

    public String getDescricao() {
        return descricao;
    }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
    public String getCor() {
        return cor;
    }
    public void setCor(String cor) {
        this.cor = cor;
    }
    public ArrayList<Aberturas> getListaDePortas() {
        return listaDePortas;
    }
    public void setListaDePortas(ArrayList<Aberturas> listaDePortas) {
        this.listaDePortas = listaDePortas;
    }
    public ArrayList<Aberturas> getListaDeJanelas() {
        return listaDeJanelas;
    }
    public void setListaDeJanelas(ArrayList<Aberturas> listaDeJanelas) {
        this.listaDeJanelas = listaDeJanelas;
    }
}

```

A primeira linha destacada é a de importação da classe **ArrayList**, cujo fichamento você fez no início dessa OT, e as chaves destacadas são as de abertura e fechamento da classe **Casa**, respectivamente. Pois bem, como já relembramos e já discutimos mais profundamente na OT anterior, a classe **Casa** tem os atributos **descricao**, **cor**, **listaDePortas** e **listaDeJanelas**. Como você acabou de implementá-la, vamos falar agora, mais detalhadamente, sobre as listas, haja vista que é o que temos de diferente, afinal, a classe **String** é sua velha conhecida.

Como você pôde aprender em seu fichamento, o **ArrayList** é uma classe que nos permite criar listas indexadas, sendo que o primeiro índice, assim como em um vetor, é **0**. Em nossa proposta, não temos controle sobre o número de portas ou janelas que serão adicionadas em nossa lista, dessa forma, é impossível que tenhamos por exemplo um vetor de tamanho *pré-definido* para o armazenamento dessas aberturas. Além disso, o **ArrayList** deixa transparente ao desenvolvedor sua manipulação (adição, remoção, consulta, substituição), que se dá de forma simples, através de seus métodos (alguns deles você também pesquisou e fichou).

Quando trabalhamos com um vetor comum, nós precisamos nos preocupar em gerenciar suas posições e não conseguimos escaloná-lo nem reduzi-lo caso precisemos de mais/menos posições futuramente. Mas isso é

possível de se fazer com o **ArrayList**! Além disso, a classe **ArrayList**, assim como a classe **LinkedList** (outra classe que nos permite criar listas e tem particularidades diferentes da **ArrayList**), implementam a interface **List**. Em seu fichamento, em algum momento você pode ter lido sobre ou até mesmo ter visto uma instância de um objeto **ArrayList** da seguinte forma:

```
List<String> listaExemplo = new ArrayList<String>();
```

Mas porque não estamos declarando nossas listas dessa forma? Se por exemplo, tivéssemos também em nossa proposta uma lista do tipo **LinkedList**, e tanto nossos **ArrayLists** quanto nossos **LinkedLists** fossem declarados como do tipo **List**, se tornariam “compatíveis”, de um mesmo tipo, e aí ganharíamos o poder de polimorfismo. Como em nossa proposta estamos trabalhando somente com o **ArrayList**, não é necessário que declaremos nossas listas dessa forma.

Dadas as explicações necessárias sobre o intuito de estarmos trabalhando com essa classe e suas particularidades, observe novamente as declarações das listas:

```
private ArrayList<Aberturas> listaDePortas = new ArrayList<Aberturas>();  
private ArrayList<Aberturas> listaDeJanelas = new ArrayList<Aberturas>();
```

A criação de um objeto **ArrayList** segue a mesma sintaxe de criação de um objeto qualquer: **Classe nomeDoObjeto = new Classe();**. No entanto, você pôde perceber que existe um elemento a mais, que é a definição do tipo da lista, entre os sinais “<” e “>”. Quando trabalhamos com o **JComboBox** na OT anterior, você já viu que definimos o tipo **<String>** para ele, e lá já mencionamos que essa definição era opcional e que voltaríamos a falar sobre essa tipagem, justamente nesse momento em que estamos agora, na definição de nossas listas.


Podemos não definir o tipo de dado a ser armazenado na lista. Isso é possível inclusive porque podemos ter objetos de diferentes tipos em uma mesma lista. Para criar um **ArrayList** sem definir o seu tipo, basta que você remova a tipagem entre os “<>”. Vamos fazer isso na classe **Casa** para a lista de portas e verificar o que aconteceria, modifique somente a linha destacada:

```
private String descricao;  
private String cor;  
private ArrayList listaDePortas = new ArrayList();  
private ArrayList<Aberturas> listaDeJanelas = new ArrayList<Aberturas>();
```

Na própria linha de declaração do **listaDePortas**, teremos um alerta...

```
Multiple markers at this line  
- ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized
```


...nos pedindo para parametrizar o tipo do **ArrayList**, pois atualmente, ele é do tipo genérico. Esse tipo genérico no Java é representado pela classe **Object**,

que é uma classe definida como “a raiz da hierarquia das classes, sendo que todas elas a implementam” . Sendo assim, se todas as classes implementam a classe **Object**, podemos dizer que todos os objetos são *objects* também, mesmo que indiretamente, certo? E é por isso que podemos não parametrizar o **ArrayList**! Mas, na verdade, quando fazemos isso, estamos dizendo que ele poderá receber qualquer *object*, ou seja, literalmente qualquer objeto de qualquer classe. Isso não é ótimo? Nem tanto... pois se a ideia fosse trabalhar com a classe **Object** “a vontade”, qual seria a utilidade das classes no paradigma de orientação a objetos? Nenhuma! Vamos precisar sim da classe **Object** em alguns momentos e você verá aplicações disso, mas nesse momento não é interessante pois sabemos exatamente o que queremos salvar em nossa lista: portas, e temos uma classe bem definida para essas portas.

Bem, **desfaça a alteração que fizemos** para continuar. Já temos a classe **Casa** e as classes **Porta** e **Janela** modeladas, com o auxílio da classe mãe **Aberturas**. Sendo assim, podemos dar continuidade à primeira funcionalidade de nosso menu. Volte então na classe **Controladora**, mais especificamente no “case 0”, correspondente a essa opção e implemente:

```
case 0:
    this.casa = new Casa();

    String descricao = EntradaSaida.solicitaDescricao("casa",0);
    String cor = EntradaSaida.solicitaCor();
    int qtdePortas = EntradaSaida.solicitaQtdeAberturas("portas");
    int qtdeJanelas = EntradaSaida.solicitaQtdeAberturas("janelas");
```

Fique calmo(a), respire ! Não temos nenhum dos 03 métodos invocados criados, nós sabemos disso e já os faremos. Agora, observe a linha que destacamos, onde a instância para a classe **Casa** é feita: até então, a classe **Controladora** possuía um atributo chamado casa, do tipo **Casa**, mas ele não possuía valor algum, tampouco era uma cópia da classe. A partir de agora, essa cópia foi realizada e no atributo casa temos disponível tudo o que está definido na classe **Casa**. Ao criarmos um objeto **Controladora**, por sua vez, não estamos imediatamente criando uma instância da classe **Casa**, concorda? Essa instância é criada somente no momento em que se faz necessária, ou seja, se efetivamente o usuário optar por criar a casa. Você consegue dizer que vantagens/desvantagens podemos ter nisso? **Discuta com seu orientador no momento da validação.**

Agora sim, na classe **EntradaSaida**, logo abaixo do último método criado anteriormente, implemente os 03 métodos que retornam as entradas de dados para os atributos da casa:


```

public static String solicitaDescricao(String descricao, int ordem) {
    if(ordem==0) {
        return JOptionPane.showInputDialog("Informe a descrição da "+descricao);
    }else {
        return JOptionPane.showInputDialog("Informe a descrição da "+ordem+" "+descricao);
    }
}

public static String solicitaCor() {
    return JOptionPane.showInputDialog("Informe a cor da casa");
}

public static int solicitaQtdeAberturas(String abertura) {
    return Integer.parseInt(JOptionPane.showInputDialog("Informe a quantidade de "+abertura));
}

```

As chaves em destaque são de abertura e fechamento de cada um dos métodos, respectivamente, para sua melhor orientação. No método **solicitaDescricao()**, estamos recebendo como parâmetro uma String **descrição** e um inteiro **ordem**, pelos quais, na chamada do método (na classe **Controladora**, no “case 0”) já passamos como argumento, respectivamente, o valor “casa” e o valor 0. Esse método nos servirá tanto para retornar a descrição da casa como de nossas portas ou janelas e para que possamos reaproveitá-lo, fizemos dessa forma. Mais adiante, você entenderá melhor como o parâmetro **ordem** funcionará. Por hora, analisando a implementação, você pode perceber que quando ele recebe o valor 0 como argumento, entende que está sendo solicitada a descrição da casa. Em relação aos métodos **solicitaCor()** e **solicitaQtdeAberturas()** entendemos que você não precisa de maiores explicações sobre sua utilidade e construção.

Muito bem, nesse exato momento, você já possui quase todas as entradas de dados para a casa realizada. Quase todas, pois ainda não preenchemos nossas listas de portas e janelas, só sabemos quantas portas e janelas deverão ser cadastradas. Vamos continuar então nossa implementação, para podermos cadastrar nossas aberturas. A linha em destaque **não** deve ser repetida, é apenas para você se orientar que ainda estamos dentro do “case 0” do nosso menu, no método **exibeMenu()** na classe **Controladora**:

```

int qtdeJanelas = EntradaSaida.solicitaQtdeAberturas("janelas");

ArrayList<Aberturas> listaDePortas = new ArrayList<Aberturas>();

for (int i=0; i<qtdePortas; i++) {
    Porta porta = new Porta();
    porta.setDescricao(EntradaSaida.solicitaDescricao("porta", (i+1)));
    porta.setEstado(EntradaSaida.solicitaEstado("porta"));
    listaDePortas.add(porta);
}

ArrayList<Aberturas> listaDeJanelas = new ArrayList<Aberturas>();

for (int i=0; i<qtdeJanelas; i++) {
    Janela janela = new Janela();
    janela.setDescricao(EntradaSaida.solicitaDescricao("janela", (i+1)));
    janela.setEstado(EntradaSaida.solicitaEstado("janela"));
    listaDeJanelas.add(janela);
}

```


Antes de discutirmos o que acabamos de implementar, vamos criar o método **solicitaEstado()** na classe **EntradaSaida**. Implemente-o logo após o fechamento do último método criado:

```
public static int solicitaEstado(String tipoAbertura) {
    String[] opcoes = {"Fechada", "Aberta"};

    return JOptionPane.showOptionDialog(null, "Informe o estado da "+tipoAbertura,
        "Estado",
        JOptionPane.DEFAULT_OPTION, JOptionPane.INFORMATION_MESSAGE, null, opcoes, opcoes[1]);
}
```

Utilizamos para esse método um **showOptionDialog()**, fornecido também pela classe **JOptionPane**. Pedimos que dê uma rápida pesquisada sobre o funcionamento desse método, não é necessário que seja realizado um fichamento sobre. Porém, gostaríamos que na validação você pudesse explicar do que se trata o último argumento repassado: **"opcoes[1]"**. Continuando, vamos recapitular o que acabamos de implementar, tendo como base a lista de portas. No *print* a seguir, temos linhas numeradas para facilitar nossa explicação e obviamente, em seu código não é necessário que elas estejam coincidindo:

```
28     ArrayList<Aberturas> listaDePortas = new ArrayList<Aberturas>();
29
30     for (int i=0; i<qtdePortas; i++) {
31         Porta porta = new Porta();
32         porta.setDescricao(EntradaSaida.solicitaDescricao("porta", (i+1)));
33         porta.setEstado(EntradaSaida.solicitaEstado("porta"));
34         listaDePortas.add(porta);
35     }
```

Na linha **28**, temos a declaração de um **ArrayList** do tipo **Aberturas**, que receberá cada uma das portas cadastradas. Como mais de uma porta pode ser cadastrada e o processo deverá se repetir, na linha **30** declaramos uma estrutura de repetição do tipo *for*, que repetirá o número de vezes informado na quantidade de portas solicitada ao usuário (para isso usamos a variável **qtdePortas**). A implementação contida nessa estrutura de repetição é a estrutura necessária para o cadastro de uma porta no **ArrayList**. Na linha **31**, criamos um novo objeto do tipo **Porta**, para onde nas linhas **32** e **33** repassaremos respectivamente os valores para seus atributos **descricao** e **estado** (os estados possíveis são 0-fechada e 1-aberta), através do uso dos métodos da classe **EntradaSaida** que retornam a descrição e o estado informados pelo usuário. É importante que se você tenha ainda alguma dúvida/estranheza com esse tipo de implementação mais “direto”, que exponha na validação para que seu orientador possa ajudá-lo. E por fim, na linha **34**, o objeto porta criado e devidamente valorado é adicionado no **ArrayList**. Todo esse processo é então repetido **qtdePortas** vezes. Você conseguiria explicar agora, a utilidade do atributo ordem do método **solicitaDescricao()**?

Vamos considerar, a título de exemplo, que em nossa casa, gostaríamos de adicionar as seguintes portas, nessa mesma ordem:

Objeto	Descrição	Estado
porta	“Da cozinha”	1
porta	“Do banheiro”	0
porta	“Do quarto”	1

Se pudéssemos então observar como nosso **ArrayList** armazenaria essas portas, seria algo como:

ArrayList<Aberturas>: listaDePortas			
	Porta: porta	Porta: porta	Porta: porta
porta.descricao	Da cozinha	Do banheiro	Do quarto
porta.estado	1	0	1
Posições	0	1	2

Como já mencionado, na primeira posição do ArrayList, a posição 0, teríamos o primeiro objeto porta, cuja descrição é “Da cozinha” e o estado tem valor 1, correspondendo a “aberto”, e assim por diante com o restante do ArrayList. Um detalhe importante a se ressaltar é que definimos que nosso **ArrayList** deve receber objetos do tipo **Aberturas**, porém, estamos adicionando objetos do tipo **Porta** nele. Podemos fazer isso pois as Portas e as Janelas são do tipo **Aberturas**, lembra? Agora, um rápido teste antes de prosseguirmos... analise o código a seguir (**não implemente-o**):

```
Aberturas porta = this.listaDePortas.get(1);
System.out.println(porta.getDescricao());
```

Você saberia dizer, considerando que o **listaDePortas** contém os objetos de nosso exemplo, o que apareceria no console?

Continuando nossa implementação do “Construir casa”, até o momento, codificamos o necessário para o usuário realizar as entradas de dados para o nosso objeto do tipo Casa, mas ainda não guardamos em nosso objeto essas entradas. É o que vamos fazer agora, e para tal, na classe Controladora, no método **exibeMenu()**, ainda no “**case 0**”, antes do **break**, implemente a linha:

```
this.casa.constroiCasa(descricao, cor, listaDePortas, listaDeJanelas);
break;
```

Obviamente após implementar essa linha, a IDE está acusando que o método **constroiCasa()** não existe na classe **Casa**. Vamos até lá implementá-lo então, após o último método declarado:

```
public void constroiCasa(String descricao, String cor, ArrayList<Aberturas> listaDePortas,
                        ArrayList<Aberturas> listaDeJanelas) {
    setDescricao(descricao);
    setCor(cor);
    setListaDePortas(listaDePortas);
    setListaDeJanelas(listaDeJanelas);
}
```

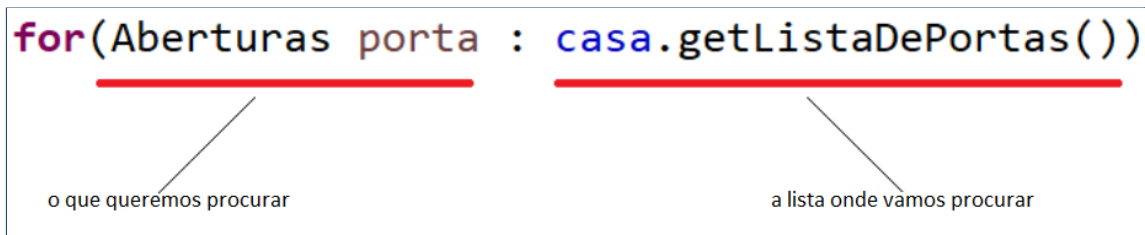
O método **constroiCasa()** é o método que finaliza com chave de ouro a criação da casa. Ainda iremos implementar a funcionalidade de movimentar as aberturas e também de ver as informações da casa. No entanto, temos um questionamento final: **você faria algo diferente no método **constroiCasa()**? É possível que sua implementação seja mais simplificada? Pense, se souber, implemente, deixando comentadas as 04 linhas que acabamos de desenvolver e descomentadas (ou seja, funcionando) as linhas de sua proposta.** Caso não chegue a nenhuma conclusão, não se preocupe, na validação você será instigado(a) a pensar mais sobre.

Para finalizar, convidamos você a realizar um teste no console para se certificar de que os atributos de nosso objeto casa foram realmente valorados. Logo após a chamada do método **constroiCasa()**, ainda no “case 0”, implemente (ignore a numeração das linhas, as usaremos nas explicações finais):

```
48      System.out.println("Descrição da casa: " + casa.getDescricao()+"\n");
49      System.out.println("Cor da casa: " + casa.getCor()+"\n");
50
51      for(Aberturas porta : casa.getListaDePortas()) {
52          System.out.println("Descrição da porta: " + porta.getDescricao()+"\n");
53          System.out.println("Estado da porta: " + porta.getEstado()+"\n");
54      }
55
56      for(Aberturas janela : casa.getListaDeJanelas()) {
57          System.out.println("Descrição da janela: " + janela.getDescricao()+"\n");
58          System.out.println("Estado da janela: " + janela.getEstado()+"\n");
59      }
60
61      break;
```

Vamos as explicações finais! Os métodos *getters()* e *setters()* já são seus conhecidos, da proposta da Calculadora, certo? Exploramos mais os métodos *setters()* do que os *getters()* lá de acordo com a situação que tínhamos. Nesse caso, para salvarmos (escrevermos) os valores nos atributos do objeto casa, utilizamos os métodos **setDescricao()**, **setCor()**, **setListaDePortas()** e **setListaDeJanelas()** e agora, para obtermos (lermos) os valores desses atributos, utilizamos os métodos *getters()* de cada um. Imaginamos que as linhas **48** e **49** não são difíceis de serem compreendidas por você. A novidade pode estar nas linhas **51** e **56**, onde estamos usando uma estrutura de repetição do tipo *for* um pouco diferente do convencional: o **for each (para cada)**. Quando temos uma lista de objetos, como é o nosso caso, podemos

percorrê-la de forma mais simplificada usando essa estrutura. Vamos entender melhor como esse **for each** funciona:



Nos atributos da classe **Casa**, temos duas listas que se forem exibidas simplesmente por seus métodos `getters()`, exibirão algo como:

`[modelo.Porta@379619aa]`. Isso acontece quando exibimos diretamente um objeto, pois seu método `toString()` é invocado, mostrando sua referência. Porém, não é isso que queremos ver, certo? Queremos ver os valores dos atributos **descrição** e **estado** dos objetos que estão dentro da nossa lista.

O **for each** pode ser dividido em duas partes, como mostra a imagem supracitada:

1. O que queremos procurar: nesse caso, as portas que estão em nossa lista, e;
2. Onde está o que queremos procurar: nesse caso, na lista de portas da classe `casa`.

Lembre-se que estamos na classe **Controladora**, e a única forma de acessar a lista de portas é pelo método `getter()` da lista, uma vez que o atributo **listaDePortas** é privado na classe **Casa**. A partir daí, podemos entender que o objeto "**Aberturas porta**" assumirá cada objeto **Porta** que for encontrado dentro da lista de portas e aí, por intermédio dele, conseguiremos exibir o que queremos: a descrição e o estado de cada porta, uma por vez. Você deve ainda estar se perguntando em que momento definimos o início/fim/incremento dessa estrutura de repetição, afinal de contas é um tipo de *for...* pois então, não precisamos fazer nada disso: o `for each` saberá que é hora de parar quando não encontrar mais objetos do tipo **Aberturas** na **listaDePortas**!

Teste sua aplicação e confira no console se todos os dados foram salvos no objeto do tipo **Casa**. Até a próxima OT 😊!