capítulo 4

Orientação a objetos básica

"Programação orientada à objetos é uma péssima idéia, que só poderia ter nascido na Califórnia."
Edsger Dijkstra -

Ao término deste capítulo, você será capaz de:

- dizer o que é e para que serve orientação a objetos,
- conceituar classes, atributos e comportamentos e
- entender o significado de variáveis e objetos na memória.

4.1 - O problema

ORIENTAÇÃO À OBJETOS

Orientação à objetos é uma maneira de programar que ajuda na organização e resolve muitos problemas enfrentados pela programação procedural.

Consideremos o clássico problema da validação de um CPF. Normalmente, temos um formulário, no qual recebemos essa informação, e depois temos que enviar esses caracteres para uma função que irá validá-lo.

Alguém te obriga a sempre validar esse CPF?

Você pode, inúmeras vezes, esquecer de chamar esse validador.

Considerando que você não erre aí, ainda temos outro problema: imagine que em algum caso, você não vá validar o CPF, ou valide de outra maneira. Por exemplo, queremos validar o CPF apenas das pessoas maiores que 18 anos. Vamos ter de colocar um `if`... mas onde? Espalhado por todo seu código...

A responsabilidade de estar verificando se o cliente tem ou não tem 18 anos, ficou espalhada por todo seu código. Seria legal poder concentrar essa responsabilidade em um lugar só, para não ter chances de esquecer isso.

Não só por isso, imagine que em algum momento precisaremos mudar essa condição... vai ter novamente de procurar todos os ifs do seu código!

Não existe uma conexão entre seus dados! Não existe uma conexão entre seus dados e suas funcionalidades! A idéia é ter essa amarra através da linguagem.

Quais as vantagens?

Orientação a objetos vai te ajudar em muito em se organizar e escrever menos, além de concentrar as responsabilidades nos pontos certos, flexibilizando sua aplicação.
Outra enorme vantagem, de onde você realmente vai economizar montanhas de código, é o **polimorfismo**, que veremos em um posterior capítulo.

4.2 - Criando um tipo

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

O que toda conta tem e é importante para nós?

- número da conta
- nome do cliente
- saldo
- limite

O que toda conta faz e é importante para nós? Isto é, o que gostaríamos de "pedir à conta".

- saca uma quantidade x
- deposita uma quantidade x
- imprime o nome do dono da conta
- devolve o saldo atual
- transfere uma quantidade x para uma outra conta y
- devolve o tipo de conta

Com isso temos o projeto de uma conta bancária. Podemos pegar esse projeto e acessar seu saldo? Não. O que temos ainda é o **projeto**. Antes precisamos **construir** uma cconta, para poder acessar o que ela tem, e pedir para ela fazer alguma coisa.

CLASSE

Ao projeto da conta, isto é, a definição da conta, damos o nome de **classe**. O que podemos construir a partir desse projeto, que são as contas de verdade, damos o nome de **objetos**.

4.3 - Uma classe em Java

Um tipo desses, como especificado acima, pode ser facilmente traduzido para Java:

```
1.  class Conta {
2.     int numero;
3.     String dono;
4.     double saldo;
5.     double limite;
6.
7.     // ..
8.
9.  }
```

String

String é uma classe em Java. Ela guarda uma palavra, isso é um punhado de caracteres. Como estamos aprendendo o que é uma classe, entenderemos melhor mais para frente.

ATRIBUTO

Por enquanto declaramos o que toda conta deve ter. Estes são os **atributos** que toda conta, quando criada, vai ter. Repare que essas variáveis foram declaradas fora de um bloco, diferente do que a gente fazia quando tinha aquele `main`. Quando uma variável é declarada diretamente dentro do escopo da classe, é chamada de variável de objeto, ou atributo.

MÉTODO

Dentro da classe, também iremos declarar o que cada conta faz, e como isto é feito. Os comportamentos que cada classe tem, isto é, o que ela faz, é chamado de **método**. Vamos começar por um simples:

```
void imprimeBanco() {
      System.out.println("Esta conta é do Banco J.");
}
```

VOID

A palavra chave `void` diz que, quando você pedir para a conta imprimir o nome do banco, nenhuma informação será enviada de volta a quem pediu.

Outra opção é sacar algum dinheiro:

```
void saca(double quantidade) {
      double novoSaldo = this.saldo - quantidade;
      this.saldo = novoSaldo;
}
```

ARGUMENTO

PARÂMETRO

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

THIS

Dentro do método, estamos declarando uma nova variável. Essa variável, assim como o argumento, vai morrer no fim do método, pois este é seu escopo. No momento que vamos acessar nosso atributo, usamos a palavra chave `this` para mostrar que esse é um atributo, e não uma simples variável.

Repare que nesse caso, a conta pode estourar o limite fixado pelo banco. Mais para frente iremos evitar essa situação, e de uma maneira muito elegante.

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

```
void deposita(double quantidade) {
      this.saldo += quantidade;
}
```

Observe que, agora, não usamos uma variável auxiliar e ainda usamos a abreviação `+=` para deixar o método bem simples.

4.4 - Criando e usando um objeto

Agora temos uma classe em Java, que especifica o que toda classe deve ter. Mas como usá-la? Além dessa, ainda teremos o **Programa.java**, e a partir dele é que iremos acessar a classe `Conta`.

NEW

Para criar (construir, instanciar) uma conta, basta usar a palavra chave `new`:

```
1.  class Programa {
2.    public static void main(String[] args) {
3.          Conta minhaConta;
4.          minhaConta = new Conta();
5.          // …
6.    }
7.  }
```

Através da variável `minhaConta` podemos acessar o objeto recém criado para alterar seu `dono`, seu `saldo` etc:

```
minhaConta.dono = "Duke";
minhaConta.saldo = 1000.0;
minhaConta.limite = 3000.0;
```

INVOCAÇÃO DE MÉTODO

É importante fixar que o ponto foi utilizado para acessar algo em `minhaConta`. Agora, `minhaConta` pertence ao Duke, tem saldo de mil reais e limite de 3 mil reais. Para mandar uma mensagem ao objeto, e pedir que ele execute um método, também usamos o ponto. O termo usado para isso é uma **invocação de método**.

O código a seguir saca um dinheiro e depois deposita outra quantia na nossa conta:

```
1.  class SacaEDeposita {
2.    public static void main(String[] args) {
3.          // criando a conta
4.          Conta minhaConta;
5.          minhaConta = new Conta();
6.
7.          // alterando os valores de minhaConta
8.          minhaConta.dono = "Duke";
9.          minhaConta.saldo = 1000;
10.         minhaConta.limite = 3000;
11.
12.         // saca 200 reais
13.         minhaConta.saca(200);
14.
15.         // deposita 500 reais
16.         minhaConta.deposita(500);
17.         System.out.println(minhaConta.saldo);
18.   }
19. }
```

Uma vez que seu saldo inicial é mil reais, se sacamos 200 reais, depositamos 500 reais e imprimimos o valor do saldo, o que será impresso?

4.5 - Métodos com retorno

Também temos um outro tipo de método, aquele que devolve o tipo de conta com base no limite:

```
boolean saca(double valor) {
      if (this.saldo < valor) {
            return false;
      }
      else {
            this.saldo = this.saldo - valor;
            return true;
      }
}
```

RETURN

Agora a declaração do método mudou! O método `saca` não tem `void` na frente, isto quer dizer que, quando acessado, ele devolve algum tipo de informação. No caso, um boolean. A palavra chave `return` indica que o método vai terminar ali, retornando tal informação.

Exemplo de uso:

```
minhaConta.saldo = 1000;
boolean consegui = minhaConta.saca(2000);
System.out.println(consegui);
```

Ou então posso eliminar a variável temporária, se desejado:

```
minhaConta.saldo = 1000;
System.out.println(minhaConta.saca(2000));
```

Meu programa pode manter na memória não só uma conta, como mais de uma:

```
Conta meuSonho;
meuSonho = new Conta();
meuSonho.saldo = 1500000;
meuSonho.limite = 1000000;
meuSonho.saca(25000);
```

4.6 - O método transfere()

E se quisermos ter um método que `transfere` dinheiro entre duas contas? Podemos fica tentados a criar um método que recebe dois parâmetros: `conta1` e `conta2` do tipo `Conta`. Mas cuidado: assim estamos pensando de maneira procedural.

A idéia é que quando chamarmos o método transfere, já teremos um objeto do tipo `Conta`, portanto o método recebe apenas **um** parâmetro do tipo, a `Conta` destino (além do valor):

```
class Conta {

	// atributos e metodos...

	void transfere(Conta destino, double valor) {
		this.saldo = this.saldo - valor;
		destino.saldo = destino.saldo + valor;
	}
}
```

Para deixar o código mais robusto, poderíamos verificar se a conta possui a quantidade a ser transferida disponível. Para ficar ainda mais interessante, você pode chamar os métodos `deposita` e `saca` já existentes para fazer essa tarefa:

```
class Conta {

	// atributos e metodos...

	boolean transfere(Conta destino, double valor) {
		boolean retirou = this.saca(valor);
		if (retirou == false) {
			// não deu pra sacar!
			return false;
		}
		else {
			destino.deposita(valor);
			return true;
		}
	}
}
```

Esse código poderia ser escrito com uma sintaxe muito mais sucinta. Como?

4.7 - Objetos são acessados por variáveis referências!

REFERÊNCIA

Quando declaramos uma variável para associar a um objeto, na verdade, essa variável não guarda o objeto, e sim uma maneira de acessá-lo, chamada de **referência**.

```
1. public static void main(String args[]) {
2.     Conta c1;
3.     c1 = new Conta();
4.     Conta c2;
```

```
5.      c2 = new Conta();
6. }
```

O correto aqui é dizer que `c1` se refere a um objeto. Não é correto dizer que `c1` é um objeto, pois `c1` é uma variável referência.

Temos agora seguinte situação:

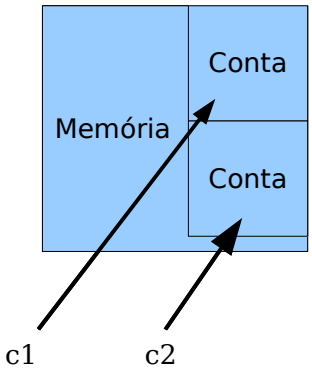
Conta c1;

c1 = new Conta();

// ...

Conta c2;

c2 = new Conta();



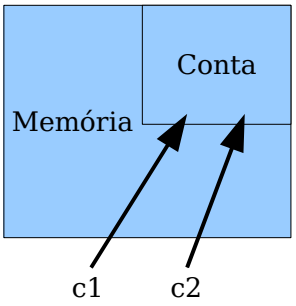

```
1.    public static void main(String args[]) {
2.     Conta c1 = new Conta();
3.     Conta c2 = c1;
4.
5.     c1.saldo = 23;
6.     System.out.println(c2.saldo);
7.    }
```

O que acontece aqui? O operador `=` copia o valor de uma variável. Mas qual é o valor da variável `c1`? É o objeto? Não. Na verdade, o valor guardado é a referência (**endereço**) para onde o objeto se encontra na memória principal.

Na memória, o que acontece nesse caso:

Conta c1 = new Conta();

Conta c2 = c1;



Quando fizemos `c2 = c1`, `c2` passa a fazer referência para o mesmo objeto que `c1` referencia nesse instante.

new

O que exatamente faz o new?

O new executa uma série de tarefas, que veremos mais adiante.

Mas, para melhor entender as referências no Java, saiba que o new, depois de alocar a memória para esse objeto, devolve uma "flecha", isto é, um valor de referência. Quando você atribui isso em uma variável, essa variável passa a se referir para esse mesmo objeto.

Podemos então ver outra situação:

```
1.    public static void main(String args[]) {
2.     Conta c1 = new Conta();
3.     c1.dono = "Duke";
4.     c1.saldo = 227;
5.
6.     Conta c2 = new Conta();
7.     c2.dono = "Duke";
8.     c2.saldo = 227;
9.
10.    if (c1 == c2) {
11.          System.out.println("Contas iguais");
12.    }
13.   }
```

O operador `==` compara o conteúdo das variáveis, mas essas variáveis não guardam o objeto, e sim o endereço em que ele se encontra. Como em cada uma dessas variáveis guardamos duas contas criadas diferentemente, eles estão em espaços diferentes da memória, o que faz o teste no `if` valer `false`. As contas podem ser equivalentes no nosso critério de igualdade, porém eles não são o mesmo. Quando se trata de objetos, pode ficar mais fácil pensar que o `==` compara se os objetos (referências na verdade) são o mesmo, e não se são iguais?

Para saber se dois objetos tem o mesmo conteúdo, você precisa comparar atributo por atributo. Veremos uma solução mais elegante para isso também.

4.8 - Continuando com atributos

As variáveis do tipo atributo, diferentemente das variáveis temporárias (declaradas dentro de um método), recebem um valor padrão. No caso numérico, valem 0, no caso de `boolean`, vale `false`.

VALORES DEFAULT

Você também pode dar **valores default**, como segue:

```
1.    class Conta {
2.     int numero = 1234;
3.     String dono = "Duke";
4.     String cpf = "123.456.789-10";
5.     double saldo = 1000;
6.     double limite = 1000;
7.    }
```

Nesse caso, quando você criar um carro, seus atributos já estão "populados" com esses valores colocados.

Seus atributos também podem ser referências para outras classes. Suponha a seguinte classe `Cliente`:

```
1.    class Cliente {
2.     String nome;
3.     String sobrenome;
4.     String cpf;
5.    }
6.
7.    class Conta {
8.     int numero;
9.     double saldo;
10.    double limite;
11.    Cliente cliente;
12.    // ..
```

```
13.   }
```

E dentro do `main` da classe de teste:

```
1.    class Teste {
2.     public static void main(String[] args) {
3.           Conta minhaConta = new Conta();
4.           Cliente c = new Cliente();
5.           minhaConta.cliente = c;
6.           // ...
7.     }
8.    }
```

Aqui simplesmente houve uma atribuição. O valor da variável `c` é copiado para o atributo `cliente` do objeto a qual `minhaConta` se refere. Em outras palavras, `minhaConta` agora tem uma referência ao mesmo `Cliente` que `c` se refere, e pode ser acessado através de `minhaConta.cliente`.

Você pode realmente navegar sobre toda essa estrutura de informação, sempre usando o ponto:

```
Cliente clienteDaMinhaConta = minhaConta.cliente;
clienteDaMinhaConta.nome = "Duke";
```

Ou ainda pode fazer isso de uma forma mais direta, e até mais elegante:

```
minhaConta.cliente.nome = "Duke";
```

NULL

Mas e se dentro do meu código eu não desse new em Cliente e tentasse acessa-lo diretamente?

```
1.    class Teste {
2.     public static void main(String[] args) {
3.           Conta minhaConta = new Conta();
4.
5.           minhaConta.cliente.nome = "paulo";
6.           // ...
7.     }
8.    }
```

Quando damos `new` em um objeto, ele o incializa com seus valores default, 0 para números, `false` para `boolean` e `null` para referências. `null` é uma palavra chave em java, que indica uma referência para nenhum objeto.

Se em algum caso você tentar acessar um atributo ou método de alguém que está se referenciando para `null`, você receberá um erro durante a execução (`NullPointerException`, que veremos mais a frente). Da para perceber então que o `new` não traz um efeito cascata, a menos que você de um valor default (ou use construtores que também veremos mais a frente):

```
1.
2.    class Conta {
3.     int numero;
4.     double saldo;
5.     double limite;
6.     Cliente cliente = new Cliente(); // quando chamarem new Conta,
7.                                      //havera um new Cliente para ele.
8.    }
```

Com esse código, toda nova `Conta` criada criado já terá um novo `Cliente` associado, sem necessidade de instanciá-lo logo em seguida da instanciação de uma `Conta`. Qual alternativa você deve usar? Depende do caso: para toda nova `Conta`

você precisa de um novo `Cliente`? É essa pergunta que deve ser respondida. Nesse nosso caso a resposta é não, mas depende do nosso problema.

4.9 - Uma Fábrica de Carros

Além do Banco que estamos criando, vamos ver como ficariam certas classes relacionadas à uma fábrica de carros. Vamos criar uma classe `Carro`, com certos atributos que descrevem suas características e com certos métodos que descrevem seu comportamento.

```
1.  class Carro {
2.   String cor;
3.   String modelo;
4.   double velocidadeAtual;
5.   double velocidadeMaxima;
6.
7.   //liga o carro
8.   void liga() {
9.         System.out.println("O carro está ligado");
10.  }
11.
12.  //acelera uma certa quantidade
13.  void acelera(double quantidade) {
14.        double velocidadeNova = this.velocidadeAtual + quantidade;
15.        this.velocidadeAtual = velocidadeNova;
16.  }
17.
18.  //devolve a marcha do carro
19.  int pegaMarcha() {
20.        if (this.velocidadeAtual < 0) {
21.              return -1;
22.        }
23.        if (this.velocidadeAtual >= 0 && this.velocidadeAtual < 40) {
24.              return 1;
25.        }
26.        if (this.velocidadeAtual >= 40 && this.velocidadeAtual < 80 {
27.              return 2;
28.        }
29.        return 3;
30.  }
31. }
```

Agora, vamos testar nosso Carro em um programa de testes:

```
1.  class TestaCarro {
2.   public static void main(String[] args) {
3.         Carro meuCarro;
4.         meuCarro = new Carro();
5.         meuCarro.cor = "Verde";
6.         meuCarro.modelo = "Fusca";
7.         meuCarro.velocidadeAtual = 0;
8.         meuCarro.velocidadeMaxima = 80;
9.
10.        // liga o carro
11.        meuCarro.liga();
12.
13.        // acelera o carro
14.        meuCarro.acelera(20);
15.        System.out.println(meuCarro.velocidadeAtual);
16.  }
17. }
```

Nosso carro pode conter também um Motor:

```
1.  class Motor {
2.   int potencia;
3.   String tipo;
4.  }
```

```
5.
6.    class Carro {
7.     String cor;
8.     String modelo;
9.     double velocidadeAtual;
10.    double velocidadeMaxima;
11.    Motor motor;
12.
13.    // ..
14.   }
```

Podemos agora criar diveros Carros e mexer com seus atributos e métodos, assim como fizemos no exemplo do Banco.

4.10 - Um pouco mais...

1) Quando declaramos uma classe, um método, ou um atributo, podemos dar o nome que quiser, seguindo uma regra. Por exemplo, o nome de um método não pode começar com um número. Pesquise sobre essas regras.

2-) Como você pode ter reparado, sempre damos nomes as variáveis com letras minúsculas. É que existem **convenções de código**, dadas pela Sun, para facilitar a legibilidade do código entre programadores. Essa convenção é *muito seguida*. Pesquise sobre ela no http://java.sun.com, procure por "code conventions".

3-) É necessário usar a palavra chave `this` quando for acessar um atributo? Para que então utilizá-la?

4-) O exercício a seguir irá pedir para modelar um "funcionário". Existe um padrão para representar suas classes em diagramas que é amplamente utilizado chamado **UML**. Pesquise sobre ele.

4.11 - Exercícios

O modelo de funcionários a seguir será utilizado para os exercícios no futuro.
É extremamente importante que o aluno faça esses exercícios para acompanhar o que é dado em aula.

O objetivo aqui é criar um sistema para gerenciar os funcionários do Banco J. Os exercícios desse capítulo são extremamente importantes.

1-) Modele um funcionário. Ele deve ter o nome do funcionário, o departamento onde trabalha, seu salário, a data de entrada no banco (`String`), seu RG (`String`), e um valor booleano que indique se o funcionário está na empresa no momento ou se já foi embora.

Você deve criar alguns métodos de acordo com o que você sentir necessidade. Além deles, crie um método `bonifica` que aumenta o `salario` do funcionário de acordo com o parâmetro passado como argumento. Crie também um método `demite` que não recebe parâmetro algum, só modifica o valor booleano indicando que o funcionário não trabalha mais aqui.

A idéia aqui é apenas modelar, isto é, só identifique que informações são importantes, e o que um funcionário faz.

2-) Transforme o modelo acima em uma classe Java. Teste-a, usando uma outra classe que tenha o `main`. Você deve criar a classe do funcionário chamada `Funcionario`, e a classe de teste você pode nomear como quiser. A de teste deve possuir o método `main`.

Um esboço da classe:

```
class Funcionario {

	double salario;

	// seus outros atributos e métodos

	void bonifica(double valor) {
		// o que fazer aqui dentro?
	}
}
```

Lembre-se de seguir a conveção java, isso é importantíssimo. Isto é, `nomeDeAtributo`, `nomeDeMetodo`, `nomeDeVariavel`, `NomeDeClasse`, etc...

Todas as classes no mesmo arquivo?

Por enquanto, você pode colocar todas as classes no mesmo arquivo, e apenas compile esse arquivo. Ele vai gerar os dois .class.

Porém é boa prática criar um arquivo .java para cada classe, e em determinados casos, você será obrigado a declarar uma classe em um arquivo separado, como veremos no capítulo 10. Isto não é importante para o aprendizado no momento.

3-) Crie um método `mostra`(), que não recebe nem devolve parâmetro algum, e simplesmente imprime todos os atributos do nosso funcionário. Dessa maneira você não precisa ficar copiando e colando um monte de `System.out.println`() para cada mudança e teste que fizer com cada um de seus funcionários, você simplesmente vai fazer:

```
Funcionario f1 = new Funcionario():
// brincadeiras com f1....
f1.mostra();
```

Veremos mais a frente o método `toString`, que é uma solução muito mais elegante para mostrar a representação de um objeto como `String`, além de não jogar tudo pro System.out (só se você desejar).

O esqueleto do método ficaria assim:

```
class Funcionario {

	// seus outros atributos e métodos

	void mostra() {
		System.out.println("Nome: " + this.nome);
		// imprimir aqui os outros atributos...
	}
}
```

4-) Construa dois funcionários com o `new`, e compare-os com o `==`. E se eles tiverem os mesmos atributos?

5-) Crie duas referências para o **mesmo** funcionário, compare-os com o `==`. Tire suas conclusões. Para criar duas referências pro mesmo funcionário:

```
Funcionario f1 = new Funcionario():
Funcionario f2 = f1;
```

6-) Em vez de utilizar uma `String` para representar a data, crie uma outra classe, chamada Data. Ela possui 3 campos int, para dia, mês e ano. Faça com que

seu funcionário passe a usá-la. (é parecido com o último exemplo, que a `Conta` passou a ter referência para um `Cliente`).

7-) O que acontece se você tentar acessar um atributo diretamente na classe? Como por exemplo:

```
Conta.saldo = 1234;
```

Esse código faz sentido? E este:

```
Conta.saca(50);
```

Faz sentido pedir para o esquema do conta sacar uma quantia?

4.12 - Desafios

1-) Um método pode chamar ele mesmo. Chamamos isso de **recursão**. Você pode resolver a série de fibonacci usando um método que chama ele mesmo. O objetivo é você criar uma classe, que possa ser usada da seguinte maneira:

```
Fibonacci fibo = new Fibonacci();
int i = fibo.calculaFibonacci(5);
System.out.println(i);
```

Aqui ira imprimir 8, já que este é o sexto número da série.

Este método `calculaFibonacci` não pode ter nenhum laço, só pode chamar ele mesmo como método. Pense nele como uma função, que usa a própria função para calcular o resultado.

2-) Porque o modo acima é extremamente mais lento para calcular a série do que o modo iterativo (que se usa um laço)?

3-) Escreva o método recursivo novamente, usando apenas uma linha. Para isso, pesquise sobre o **operador condicional ternário**. (ternary operator)

4.13 - Fixando o conhecimento

O objetivo dos exercícios a seguir é fixar o conceito de classes e objetos, métodos e atributos. Dada a estrutura de uma classe, basta traduzi-la para a linguagem Java e fazer uso de um objeto da mesma em um programa simples.

Se você está com dificuldade em alguma parte desse capítulo, aproveite e treine tudo o que vimos até agora nos pequenos programas abaixo:

Programa 1

Classe: Pessoa.
Atributos: nome, idade.
Método: void fazAniversario()

Crie uma pessoa, coloque seu nome e idade inicial, faça alguns aniversários (aumentando a idade) e imprima seu nome e sua idade.

Programa 2

Classe: Porta
Atributos: aberta, cor, dimensaoX, dimensaoY, dimensaoZ

Métodos: void abre(), void fecha(),
void pinta(String s), boolean estaAberta()

Crie uma porta, abra e feche a mesma, pinte-a de diversas cores, altere suas dimensões e use o metodo `estaAberta` para verificar se ela esta aberta.

Programa 3

Classe: Casa
Atributos: cor, porta1, porta2, porta3
Método: void pinta(String s),
int quantasPortasEstaoAbertas()

Crie uma casa e pinte-a. Crie três portas e coloque-as na casa; abra e feche as mesmas como desejar. Utilize o método `quantasPortasEstaoAbertas` para imprimir o número de portas abertas.