04

Aprendendo design patterns através dos candlesticks japoneses

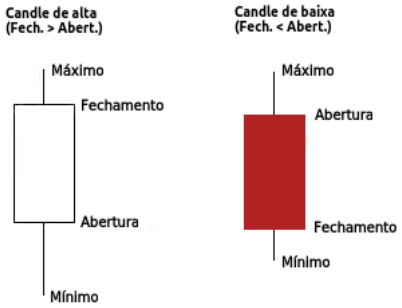
Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/laboratorio-java/stages/02/capitulo2.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Candlestick: O Japão e o arroz.

Yodoya Keian era um mercador japonês do século XVII. Ele rapidamente se tornou muito rico, dadas as suas habilidades de transporte e precificação do arroz, uma mercadoria em crescente produção e consumo no país. Sua situação social de mercador não permitia que ele fosse tão rico dado o sistema de castas da época e, logo, o governo confiscou todo seu dinheiro e suas posses.

Apesar da triste história, foi em Dojima, no jardim do próprio Yodoya Keian, que nasceu a bolsa de arroz do Japão. Lá eram negociados, precificados e categorizados vários tipos de arroz. Para anotar os preços do arroz, desenhavam-se figuras no papel. Essas figuras parecem muito com velas -- daí a analogia **candlestick** (*candle*, que do inglês significa vela e *candlestick*, que significa castiçal).

Um *candlestick* indica 4 valores: o maior e o menor preço do dia (as pontas), o primeiro e o último preço do dia (conhecidos como abertura e fechamento, respectivamente).

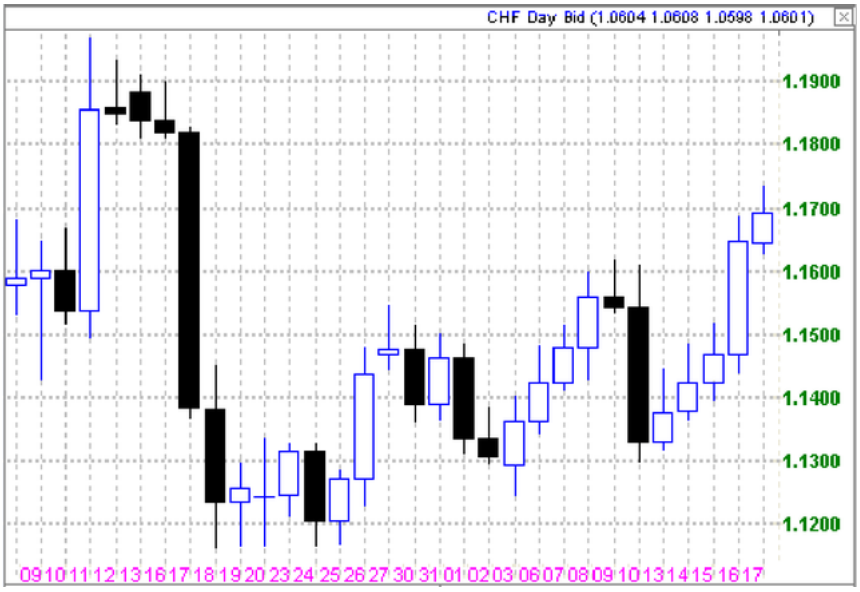


Os preços de abertura e fechamento são as linhas horizontais e dependem do tipo de *candle*: se for de alta, o preço de abertura é embaixo; se for de baixa, é em cima. Um *candle* de alta costuma ter cor azul ou branca e os de baixa costumam ser vermelhos ou pretos. Caso o preço não tenha se movimentado, o *candle* tem a mesma cor que a do dia anterior.

Para calcular as informações necessárias para a construção de um *candlestick*, são necessários os dados de todas as **negociações** (*trades*) de um dia. Uma **negociação**, como já vimos, possui três informações: o **preço** pelo qual foi comprado, a **quantidade** de ativos comprados e a **data** em que ela foi executada.

Os *Candlesticks* servem para **resumir graficamente as principais informações sobre as negociações daquele dia**, tornando estes dados fáceis de visualizar.

Para você ter uma noção de como é o gráfico de *Candlesticks* ao longo de um dia:



Apesar de falarmos que o *candlestick* representa os principais valores de **um dia**, ele pode ser usado para os mais variados intervalos de tempo: um *candlestick* pode representar 15 minutos, ou uma semana, dependendo se você está analisando o ativo para curto, médio ou longo prazo.

Modelando o sistema

Como se trata de um analisador gráfico, além da classe negociação que já modelamos, vamos ter que modelar outros objetos do nosso sistema, entre eles:

- **Candlestick**: guardando as informações do *candle*, além do volume de dinheiro negociado.
- **SérieTemporal**: que guarda um conjunto de *candle* de um determinado período de tempo.

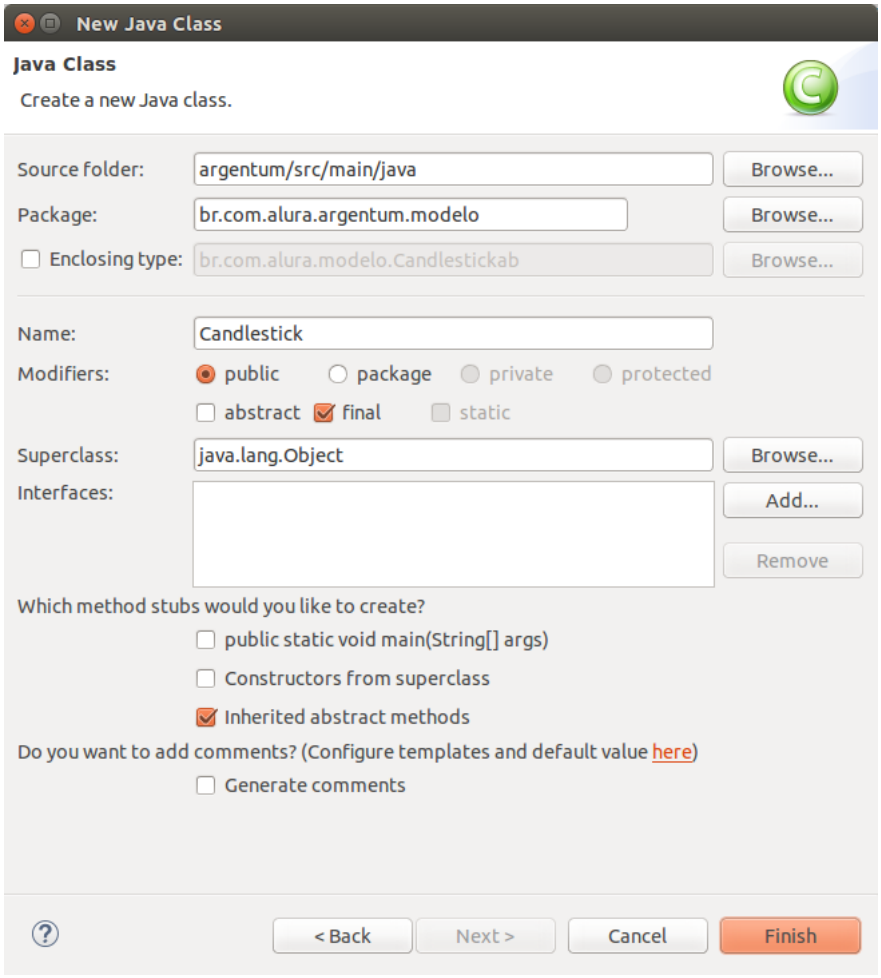
Essas classes, junto da `Negociacao` são a base do projeto **Argentum** que criaremos durante o treinamento.

As funcionalidades serão as seguintes:

- Resumir **negociações** em *candlesticks*. Nossa base serão as negociações. Precisamos converter uma lista de negociações em uma lista de *candles*.
- Converter *candlesticks* em uma **série temporal**. Dada uma lista de *candles*, precisamos criar uma série temporal.
- Utilizar indicadores técnicos. Para isso, implementar um pequeno *framework* de indicadores e criar alguns deles de formas a facilitar o desenvolvimento de novos.
- Gerar gráficos Embutíveis e interativos na interface gráfica em Java, dos indicadores que criamos.

Criando nossa classe Candlestick

Vamos criar nossa classe `Candlestick` , baseado no que aprendemos. Utilize o atalho **CTRL + N** e busque por *Class*, marque-a como `final` já que , como aprendemos capítulo passado, a classe `Candlestick` também será **imútavel**:

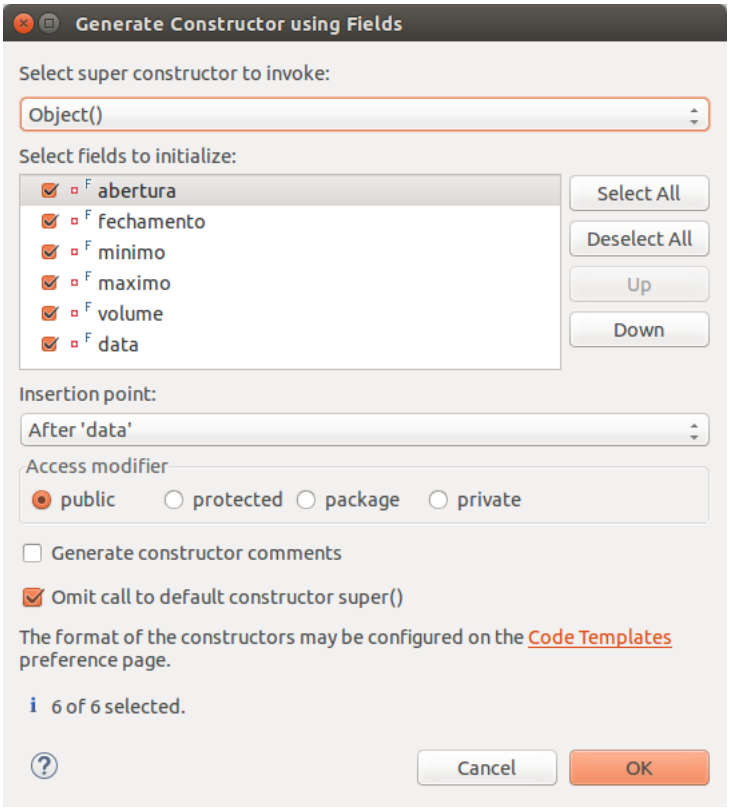


Vamos criar os seguintes atributos **finais**:

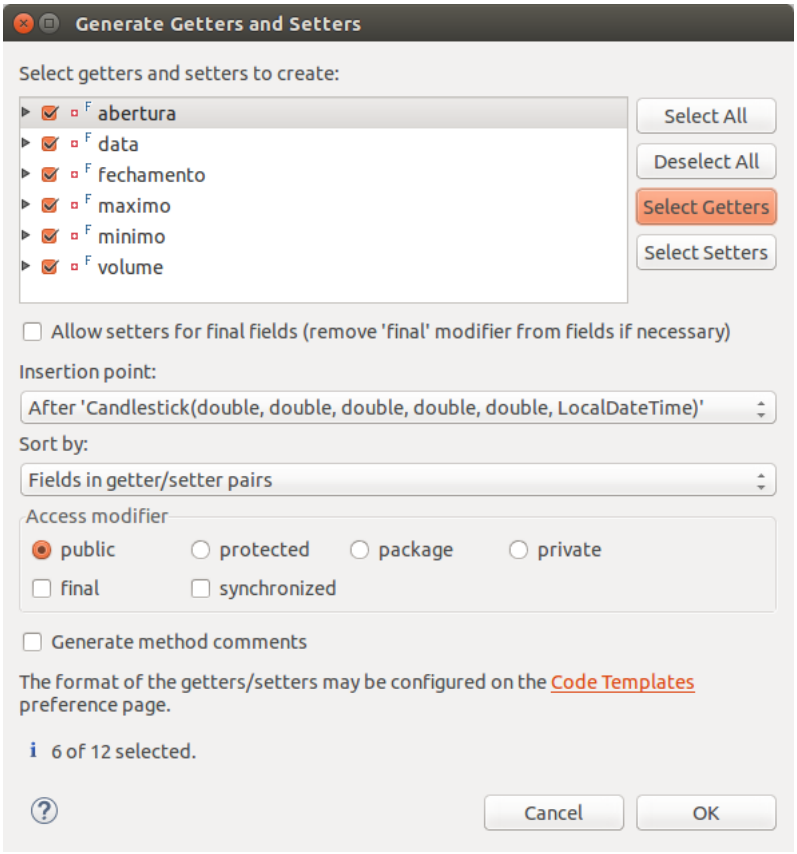
```
public final class Candlestick {

    private final double abertura;
    private final double fechamento;
    private final double minimo;
    private final double maximo;
    private final double volume;
    private final LocalDateTime data;
}
```

Agora podemos usar o atalho **CTRL + 3** e buscar por *GCUF*, para criar o construtor:



Aproveite também para gerar os seis respectivos getters com **CTRL + 3** e *GGAS*:



Sua classe final deve ficar assim:

```
public final class Candlestick {

    private final double abertura;
    private final double fechamento;
    private final double minimo;
```

```
    private final double maximo;
    private final double volume;
    private final LocalDateTime data;

    public Candlestick(double abertura, double fechamento, double minimo,
            double maximo, double volume, LocalDateTime data) {
        this.abertura = abertura;
        this.fechamento = fechamento;
        this.minimo = minimo;
        this.maximo = maximo;
        this.volume = volume;
        this.data = data;
    }

    public double getAbertura() {
        return abertura;
    }

    public double getFechamento() {
        return fechamento;
    }

    public double getMinimo() {
        return minimo;
    }

    public double getMaximo() {
        return maximo;
    }

    public double getVolume() {
        return volume;
    }

    public LocalDateTime getData() {
        return data;
    }
}
```

Pronto! Agora modelamos também a nossa classe `Candlestick` , nosso sistema está começando a ganhar forma.

Adicionando alguns métodos às nossas classes

Por enquanto, tanto a nossa classe `Candlestick` quanto a classe `Negociacao` só possuem simples *getters*. Vamos adicionar alguns métodos a elas, para nos trazer algumas informações que serão úteis no futuro.

Primeiramente, na nossa classe *Negociacao*, sabemos que um dado importante que diz muito sobre a estabilidade de uma ação na bolsa de valores é o **volume** de dinheiro negociado em um período. Vamos fazer nossa classe `Negociacao` devolver o volume do dinheiro transferido naquela negociação. Na prática, basta multiplicar o **preço** pago pela **quantidade** de ações negociadas, resultando no total de dinheiro que aquela negociação realizou.

Adicione o método `getVolume` na classe `Negociacao` :

```
public class Negociacao {

    // restante do código

    public double getVolume(){
        return preco * quantidade;
    }
}
```

Repare que um método simples, que parece ser um simples *getter* pode (e deve muitas vezes) encapsular **regras de negócio que não necessariamente refletem um atributo da classe**.

Agora na nossa classe `Candlestick`, podemos criar dois métodos de negócio, para que o `Candlestick` nos diga se é do tipo de alta ou de baixa:

```
public class Candlestick {

    // restante do código

    public boolean isAlta(){
        return this.abertura < this.fechamento;
    }

    public boolean isBaixa(){
        return this.abertura > this.fechamento;
    }
}
```

Pronto, mais dois métodos que nos trazem informações relevantes sobre o nosso modelo.

Resumo diário das Negociações

Agora que temos as classes que representam negociações na bolsa de valores (`Negociacao`) e resumos diários dessas negociações (`Candlestick`), falta apenas fazer a ação de resumir as negociações de um dia em uma *candle*.

A regra é um tanto simples: dentre uma lista de negociações, precisamos descobrir quais são os valores a preencher na `Candlestick` :

- **Abertura**: preço da primeira negociação do dia;
- **Fechamento**: preço da última negociação do dia;
- **Mínimo**: preço da negociação mais barata do dia;
- **Máximo**: preço da negociação mais cara do dia;
- **Volume**: quantidade de dinheiro que passou em todas as negociações nesse dia;
- **Data**: a qual dia o resumo se refere.

Algumas dessas informações são fáceis de encontrar por que temos uma **convenção** no sistema: quando vamos criar a *candle*, a lista de negociações já vem ordenada por tempo. Dessa forma, a abertura e o fechamento são triviais: basta recuperar o preço , respectivamente, da primeira e da última negociação do dia!

Já mínimo, máximo e volume precisam que todos os valores sejam verificados. Dessa forma, precisamos passar por cada negociação da lista, verificando se aquele valor é menor do que todos os outros que já vimos, maior que nosso máximo atual. Aproveitando esse processo de passar por cada negociação, já vamos somando o volume de cada negociação.

O algoritmo, agora, está completamente especificado! Basta passarmos essas ideias para código.

Em que classe colocar?

Falta apenas, antes de pôr em prática o que aprendemos, decidirmos onde vai esse código de criação de *candlestick*. Pense bem a respeito disso: será que uma negociação deveria saber resumir vários de si em uma *candle*? Ou será que uma *candlestick* deveria saber gerar um objeto do próprio tipo *candlestick* a partir de uma lista de negociações.

Em ambos os cenários, nossos modelos têm que ter informações a mais que, na realidade, são responsabilidades que não cabem a eles!

Criaremos, então, uma classe que: *dado a matéria-prima, nos constrói uma candle*. E uma classe com esse comportamento, que recebem o necessário para criar um objeto e **encapsulam** o algoritmo para tal criação, costuma ser chamadas de **Factory**.

No nosso caso particular, essa é uma fábrica que cria *candlesticks*, então, seu nome será `CandlestickFactory`.

Perceba que esse nome, apesar de ser um *design Pattern*, nada mais faz do que encapsular uma lógica um pouco mais complexa, isto é, apenas aplica boas práticas de orientação a objetos.

Criando nossa fábrica de Candlesticks

Queremos, dada uma lista de negociações de um dia específico e já em ordem de data, devolver um *candlestick* com os dados daquele dia.

Vamos começar criando a classe, no pacote `br.com.caelum.argentum.modelo`:

```
public class CandlestickFactory {

}
```

É um método que faz exatamente o que queremos, retornar um *candlestick* a partir da **lista de negociações** do **dia**. Para isso vamos dar um nome bem expressivo para ele:

```
public class CandlestickFactory {

    // Nosso método vai retornar um candlestick
    // E espera receber uma lista de negociações e uma data
    public Candlestick constroiCandleParaData(List<Negociacao> negociacoes,
            LocalDateTime data) {

    }
}
```

Como ele retorna um *candlestick*, já podemos adicionar seu retorno também:

```
public class CandlestickFactory {

    public Candlestick constroiCandleParaData(List<Negociacao> negociacoes, LocalDateTime data)

        // Já podemos criar o retorno com os valores que o candlestick espera no construtor
        return new Candlestick(abertura, fechamento, minimo, maximo, volume, data);
    }
}
```

Agora vamos descobrir o valor de cada uma das variáveis que farão parte do nosso *candlestick*. Os valores de **abertura** e **fechamento** são fáceis, já que por **convenção** do nosso sistema nós recebemos uma lista ordenada por data, basta pegar o valor do primeiro e último elementos:

```
public class CandlestickFactory {

    public Candlestick constroiCandleParaData(List<Negociacao> negociacoes, LocalDateTime data)

        // Os valores e fechamento são a primeira e última negociação da lista, respectivamente
        double abertura = negociacoes.get(0).getPreco();
        double fechamento = negociacoes.get(negociacoes.size() - 1).getPreco();

        return new Candlestick(abertura, fechamento, minimo, maximo, volume, data);
    }
}
```

Também precisamos do **volume** de negociações do dia, e para conseguir isso precisamos obter o volume de cada **negociação** da lista e ir adicionando cada uma delas a uma variável **volume**. Vamos primeiro criar nossa variável volume:

```
public class CandlestickFactory {

    public Candlestick constroiCandleParaData(List<Negociacao> negociacoes, LocalDateTime data)

        double abertura = negociacoes.get(0).getPreco();
        double fechamento = negociacoes.get(negociacoes.size() - 1).getPreco();

        // Criando variável volume
        double volume = 0;

        return new Candlestick(abertura, fechamento, minimo, maximo, volume, data);
    }

}
```

E agora percorrer a lista de negociações, vamos iterando-a com um `foreach`, e adicionando os valores na nossa variável recém criada:

```
public class CandlestickFactory {

    public Candlestick constroiCandleParaData(List<Negociacao> negociacoes, LocalDateTime data)
```

```
        double abertura = negociacoes.get(0).getPreco();
        double fechamento = negociacoes.get(negociacoes.size() - 1).getPreco();

        double volume = 0;

        // Iterando sobre cada negociação da lista para atualizar a variável volume
        for (Negociacao negociacao : negociacoes) {
            volume += negociacao.getVolume();
        }

        return new Candlestick(abertura, fechamento, minimo, maximo, volume, data);
    }
}
```

Agora falta apenas o **máximo** e **mínimo**, que são os valores do preço da maior e da menor negociação do dia. Primeiro vamos criar duas variáveis `double` , `maximo` e `mínimo` , e vamos chutar um valor para elas, por exemplo o valor da primeira negociação:

```
public class CandlestickFactory {

    public Candlestick constroiCandleParaData(List<Negociacao> negociacoes, LocalDateTime data)

        double abertura = negociacoes.get(0).getPreco();
        double fechamento = negociacoes.get(negociacoes.size() - 1).getPreco();

        // Criando variáveis máximo e mínimo com os valores da primeira negociação.
        double minimo = negociacoes.get(0).getPreco();
        double maximo = negociacoes.get(0).getPreco();

        double volume = 0;

        for( Negociacao negociacao : negociacoes ){
            volume += negociacao.getVolume();
        }

        return new Candlestick(abertura, fechamento, minimo, maximo, volume, data);
    }

}
```

Em seguida, podemos comparar o valor de **cada** `negociacao` da lista com o da variável `minimo` , se encontrarmos um valor menor do que o atual basta substituirmos. Podemos adotar essa mesma estratégia para o valor máximo, e como queremos comparar com todas as negociações da lista, já aproveitaremos o `foreach` que utilizamos para calcular o volume anteriormente, ficando tudo no final desta forma:

```
public class CandlestickFactory {

    public Candlestick constroiCandleParaData(List<Negociacao> negociacoes, LocalDateTime data)

        double abertura = negociacoes.get(0).getPreco();
        double fechamento = negociacoes.get(negociacoes.size() - 1).getPreco();
```

```
        double minimo = negociacoes.get(0).getPreco();
        double maximo = negociacoes.get(0).getPreco();

        double volume = 0;

        for( Negociacao negociacao : negociacoes ){
            volume += negociacao.getVolume();

            // Se o valor da negociação da iteração atual for maior, atribuímos este valor a vai
            if (negociacao.getPreco() > maximo) {
                maximo = negociacao.getPreco();
            // A mesma coisa para o mínimo
            } else if (negociacao.getPreco() < minimo) {
                minimo = negociacao.getPreco();
            }
        }

        return new Candlestick(abertura, fechamento, minimo, maximo, volume, data);
    }

}
```

Agora que já escrevemos uma boa quantidade de código, vamos testar se tudo funciona criando 4 negociações e calculando o *candlestick*, finalmente.

Crie uma classe chamada **TestaCandlestickFactory** no pacote **br.com.alura.argentum.testes**:

```
public class TesteCandlestickFactory {

}
```

E dentro dessa classe crie o método `main`, que você pode apenas escrever *main* e utilizar o **CTRL + Espaço** para autocompletar:

```
public class TesteCandlestickFactory {

    public static void main(String[] args) {

    }
}
```

Agora, como precisamos criar algumas negociações para testar nossa `CandlestickFactory`, vamos aproveitar essa oportunidade para ver como trabalhamos com a classe `LocalDateTime`.

Para obtermos a data de hoje, utilizamos o método `.now()`, ficando assim;

```
public class TesteCandlestickFactory {

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

```
        LocalDateTime hoje = LocalDateTime.now();
    }
}
```

Conseguindo a data , podemos criar 4 novas negociações:

```
public class TesteCandlestickFactory {

    public static void main(String[] args) {

        LocalDateTime hoje = LocalDateTime.now();

        Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
        Negociacao negociacao2 = new Negociacao(37.0, 100, hoje);
        Negociacao negociacao3 = new Negociacao(45.5, 100, hoje);
        Negociacao negociacao4 = new Negociacao(42.5, 100, hoje);

    }
}
```

Com as negociações instanciadas, podemos adicioná-las numa lista através do método `asList()` da classe Arrays:

```
public class TesteCandlestickFactory {

    public static void main(String[] args) {

        LocalDateTime hoje = LocalDateTime.now();

        Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
        Negociacao negociacao2 = new Negociacao(37.0, 100, hoje);
        Negociacao negociacao3 = new Negociacao(45.5, 100, hoje);
        Negociacao negociacao4 = new Negociacao(42.5, 100, hoje);

        List<Negociacao> negociacoes = Arrays.asList(negociacao1, negociacao2,
                negociacao3, negociacao4);

    }
}
```

Este método de `Arrays` aceita *varargs*, o que nos possibilita invocá-lo **passando os elementos do array separados por vírgulas**. Algo parecido com um *autoboxing* de arrays.

Com a lista de negociações e a data nas mãos, podemos testar a nossa `CandlestickFactory` , vamos criá-la e imprimir o *candlestick* no final para confirmar se tudo ocorreu corretamente:

```
public class TestaCandlestickFactory {

    public static void main(String[] args) {

        LocalDateTime hoje = LocalDateTime.now();
```

```
        Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
        Negociacao negociacao2 = new Negociacao(37.0, 100, hoje);
        Negociacao negociacao3 = new Negociacao(45.5, 100, hoje);
        Negociacao negociacao4 = new Negociacao(42.5, 100, hoje);

        List<Negociacao> negociacoes = Arrays.asList(negociacao1, negociacao2,
                negociacao3, negociacao4);

        CandlestickFactory fabrica = new CandlestickFactory();
        Candlestick candle = fabrica.constroiCandleParaData(negociacoes, hoje);

        System.out.println(candle.getAbertura());
        System.out.println(candle.getFechamento());
        System.out.println(candle.getMinimo());
        System.out.println(candle.getMaximo());
        System.out.println(candle.getVolume());

    }

}
```

A sua saída deve ser algo parecido com isso:

```
40.5
42.5
37.0
45.5
16550.0
```

Conseguimos testar nossa `CandlestickFactory` e ela parece estar funcionando corretamente, mas será que esse é o melhor jeito de se testar em Java? No próximo capítulo abordaremos melhor este assunto!

O que aprendemos:

- O que é um *candlestick* e a sua história.
- A modelar a classe `Candlestick` .
- O atalho **CTRL + N** para abrir o menu de *Novo...* no Eclipse.
- Nem sempre as regras de um negócio refletem em atributos de uma classe.
- Como resumir nossas negociações em *candlestick*.
- O *design pattern* **Factory** e como criar uma *factory* de *candlesticks*.
- Como utilizar na prática o `LocalDateTime` do Java 8.
- Como criar um lista usando o método `asList()` de `Arrays` .