CAPÍTULO 9

Gráficos interativos com Primefaces

"A única pessoa educada é aquela que aprendeu a aprender e a mudar."
— Carl Rogers

9.1 - POR QUE USAR GRÁFICOS?

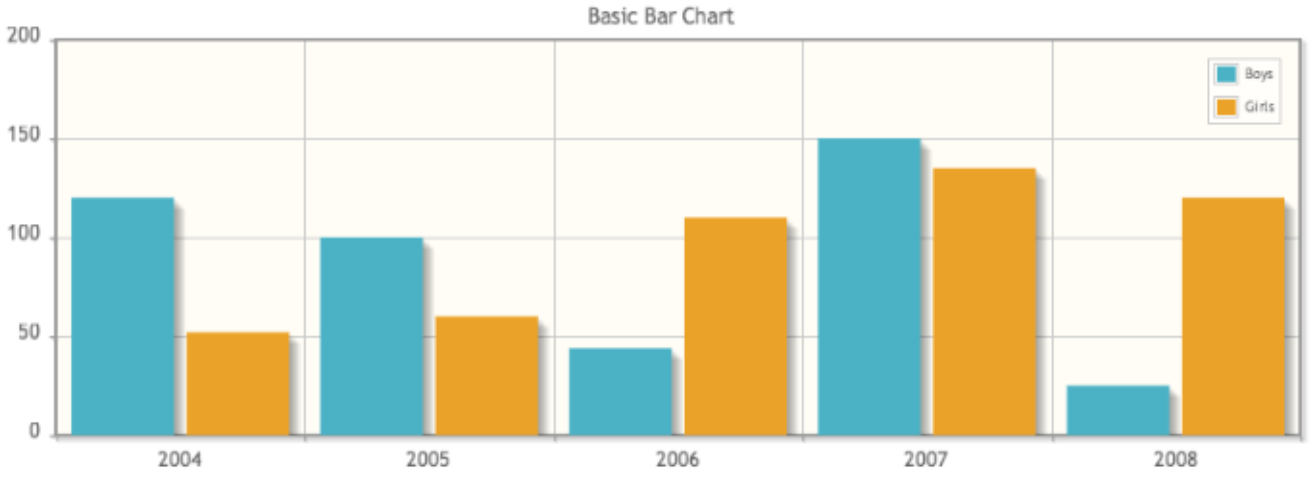
Nossa aplicação apresenta os dados das negociações tabularmente através do componente `p:dataTable`. Essa forma de mostrar informações é interessante para analisarmos dados um a um, mas não ajudam muito quando queremos ter uma ideia do que acontece com dados coletivamente.

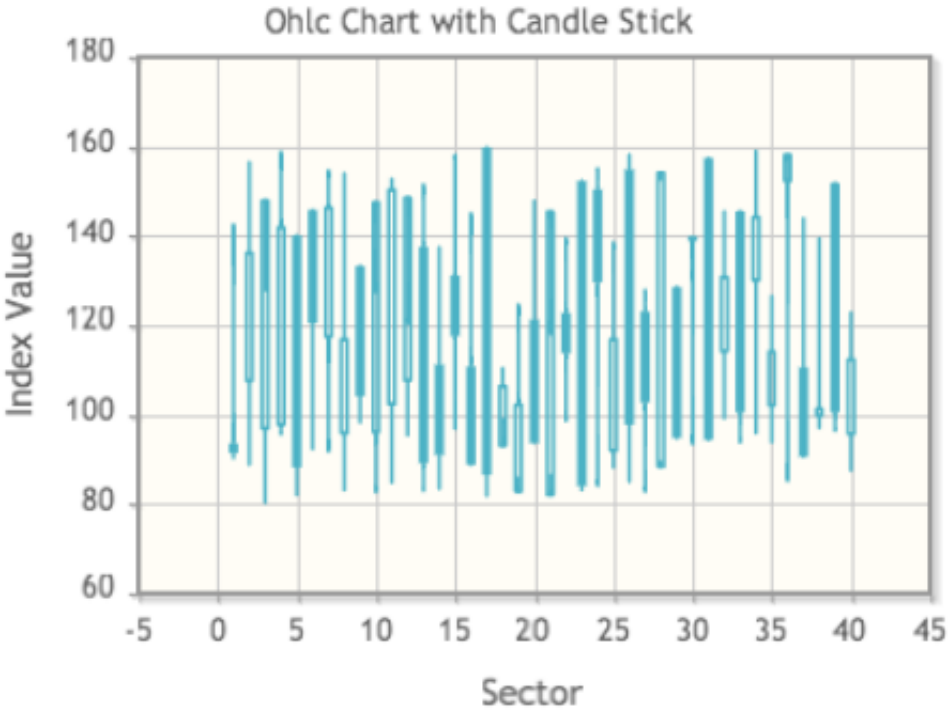
Gráficos comprovadamente ajudam no entendimento mais abrangente dos dados e são mais fáceis de analisar do que números dentro de uma tabela. É simples reconhecer padrões de imagens, por exemplo na análise técnica de valores da bolsa.

Para o projeto `Argentum`, apresentaremos os valores da `SerieTemporal` em um gráfico de linha, aplicando algum indicador como abertura ou fechamento. Continuaremos com a biblioteca Primefaces que já vem com suporte para vários tipos de gráficos.

Exemplos de gráficos

O Primefaces já possui diversos componentes para gráficos: é possível utilizá-lo para desenhar gráficos de linha, de barra, de pizza, de área, gráficos para atualização dinâmica e até para Candles, entre outros. Também é possível exportar e animar gráficos.





9.2 - Gráficos com o Primefaces

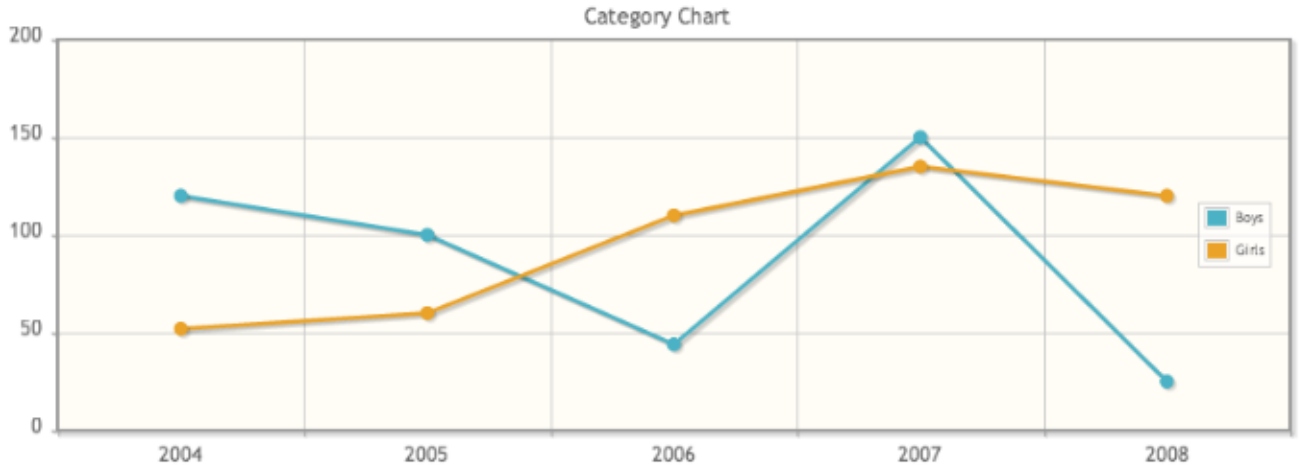
Vamos usar o Primefaces para gerar um gráfico que mostra a evolução dos valores da série.

Nosso projeto está configurado e já podemos decidir qual gráfico utilizar. Para facilitar a decisão e ao mesmo tempo ver as possibilidades e tipos de gráficos disponíveis, o showcase do Primefaces nos ajudará muito:

http://www.primefaces.org/showcase/index.xhtml

Nele encontramos o resultado final e também o código utilizado para a renderização. Vamos programar usando o componente `p:chart` que deve mostrar

os valores de abertura ou de fechamento da `SerieTemporal`.



O uso de componente é simples, veja o código de exemplo do showcase:

```
<p:chart type="line" model="#{chartView.lineModel2}" style="height:300px;"/>
```

Podemos ver que o componente recebe os dados (`model`) através da *Expression Language* que chama o *Managed Bean* `#{chartView.lineModel2}`.

9.3 - Documentação

A documentação do Primefaces está disponível na internet e na forma de um guia do usuário em PDF. Ela fará parte do dia-a-dia do desenvolvedor, que a consultará sempre que necessário:

http://www.primefaces.org/documentation.html

Também há o tradicional Javadoc disponível em:

http://www.primefaces.org/docs/api/5.1/. Devemos usar ambos para descobrir funcionalidades e propriedades dos componentes.

No showcase também há um exemplo do uso do *Managed Bean*, porém para maiores informações é fundamental ter acesso ao Javadoc e à documentação da biblioteca para saber quais classes, atributos e métodos utilizar.

> **JSF e CSS**
>
> O Primefaces ajuda o desenvolvedor através de seu suporte a temas, mas nem sempre ele é suficiente, inclusive quando há uma equipe de designers responsável em determinar a apresentação da aplicação.
>
> Apesar de trabalharmos com componentes, no final o que é enviado para o cliente é puro HTML. Assim, quem conhece CSS pode aplicar estilos para sobrescrever o visual da aplicação.
>
> A Caelum oferece o curso **Desenvolvimento Web com HTML, CSS e JavaScript**, para aqueles que querem aprender a criar interfaces Web com experiência rica do usuário, estruturação correta e otimizações SEO.

Aplicando ao nosso projeto

Para mostrarmos o gráfico no Argentum, usaremos a tag do Primefaces vista acima:

```
<p:chart type="line" />
```

Da forma como está, no entanto, não há nada a ser mostrado no gráfico. Ainda falta indicarmos para o componente que os dados, o modelo do gráfico, será disponibilizado por nosso `ManagedBean`. Em outras palavras, faltou indicarmos que o `model` desse gráfico será produzido em `argentumBean.modeloGrafico`.

Nossa adição ao `index.xhtml` será, portanto:

```
<p:chart type="line" model="#{argentumBean.modeloGrafico}" />
```

9.4 - Definição do modelo do gráfico

Já temos uma noção de como renderizar gráficos através de componentes do

Primefaces. O desenvolvedor não precisa se preocupar como detalhes de JavaScript, imagem ou animações. Tudo isso é encapsulado no próprio componente, seguindo boas práticas do mundo orientado a objetos.

Apenas, será necessário informar ao componente quais são os dados a serem plotados no gráfico em questão. Esses dados representam o modelo do gráfico e devem ser disponibilizados como o `model` para o `p:chart` em um objeto do tipo `org.primefaces.model.chart.ChartModel`.

Essa é a classe principal do modelo e há classes filhas especializadas dela como `LineChartModel`, `PieChartModel` ou `BubbleChartModel`. No JavaDoc podemos ver todas as filhas da `ChartModel` e ainda no *showcase* é possível ver o `ManagedBean` responsável por cada modelo de gráfico. Assim, saberemos qual delas devemos usar de acordo com o gráfico escolhido:

Javadoc: http://www.primefaces.org/docs/api/5.1/org/primefaces/model/chart/ChartModel.html

No nosso projeto, utilizaremos o `LineChartModel`, já que queremos plotar pontos em um gráfico de linha. Um `LineChartModel` recebe uma ou mais `ChartSeries` e cada `ChartSeries` representa uma linha no gráfico do componente `p:chart`. Uma `ChartSeries`, por sua vez, contém todos os pontos de uma linha do gráfico, isto é, os valores X e Y que serão ligados pela linha do gráfico.

Vejo como fica o código fácil de usar:

```
ChartSeries serieGrafico = new ChartSeries("Abertura");
serieGrafico.set("dia 1", 20.9);
serieGrafico.set("dia 2", 25.1);
serieGrafico.set("dia 3", 22.6);
serieGrafico.set("dia 4", 24.6);

LineChartModel modeloGrafico = new LineChartModel();
modeloGrafico.addSeries(serieGrafico);
```

Uma `ChartSeries` recebe no construtor a legenda da linha que ela representa (*label*) e, através do método `set`, passamos os valores de cada ponto nos eixos horizontal e vertical, respectivamente. No exemplo acima, os valores colocados são fixos, mas na nossa implementação do Argentum, é claro, iteraremos pela `SeriaTemporal` calculando os indicadores sobre ela.

Isto é, uma vez que temos a `SerieTemporal`, nosso código para plotar a média móvel simples do fechamento em uma `ChartSeries` será semelhante a este:

```
SerieTemporal serie = ...
ChartSeries chartSeries = new ChartSeries("MMS do Fechamento");
MediaMovelSimples indicador = new MediaMovelSimples();
for (int i = 2; i < serie.getUltimaPosicao(); i++) {
  double valor = indicador.calcula(i, serie);
  chartSeries.set(i, valor);
}

LineChartModel modeloGrafico = new LineChartModel();
modeloGrafico.addSeries(chartSeries);
```

Conseguir a série temporal é um problema pelo qual já passamos antes. Relembre que a `SerieTemporal` é apenas um *wrapper* de uma lista de `Candles`. E a lista de `Candles` é gerada pelo `CandlestickFactory` resumindo uma lista com muitas `Negociacoes`.

Se procurarmos o local no nosso código onde pegamos a lista de negociações do web service, vamos notar que isso acontece no construtor da `ArgentumBean`. Seu código completo ficaria assim;

```
public ArgentumBean() {
  this.negociacoes = new ClienteWebService().getNegociacoes();
  List<Candle> candles = new CandleFactory().constroiCandles(negociacoes);
  SerieTemporal serie = new SerieTemporal(candles);

  ChartSeries chartSeries = new ChartSeries("MMS do Fechamento");
  MediaMovelSimples indicador = new MediaMovelSimples();
  for (int i = 2; i < serie.getUltimaPosicao(); i++) {
    double valor = indicador.calcula(i, serie);
    chartSeries.set(i, valor);
  }

  this.modeloGrafico = new LineChartModel();
  modeloGrafico.addSeries(chartSeries);
}
```

Note, no entanto, que esse trecho de código está com responsabilidades demais: ele busca as negociações no *web service*, cria a série temporal, plota um indicador e disponibiliza o modelo do gráfico. E, pior ainda, esse código ainda está no construtor do `ArgentumBean`!

Com todo esse código no construtor do bean, teríamos um código mais sujo, pouco coeso e muito difícil de testar. O que acontece aqui é que não estamos separando responsabilidades o bastante e nem encapsulando a lógica de geração de gráfico corretamente.

9.5 - Isolando a API do Primefaces: baixo acoplamento

O que acontecerá se precisarmos criar dois gráficos de indicadores diferentes? Vamos copiar e colar todo aquele código e modificar apenas as partes que mudam? E se precisarmos alterar algo na geração do modelo? Essas mudanças não serão fáceis se tivermos o código todo espalhado pelo nosso programa.

Os princípios de orientação a objetos e as boas práticas de programação vêm ao nosso socorro aqui. Vamos **encapsular** a maneira como o modelo do gráfico é criado na classe `GeradorModeloGrafico`.

Essa classe deve ser capaz de gerar o `ChartModel` para nosso gráfico de linhas, com os pontos plotados pelos indicadores com base nos valores de uma série temporal. Isto é, nosso `GeradorModeloGrafico` precisa receber a `SerieTemporal` sobre a qual plotar os indicadores.

Se quisermos restringir o gráfico a um período menor do que o devolvido pelo *web service*, é uma boa recebermos as posições de início e fim que devem ser plotadas. Esse intervalo também serve para que não tentemos calcular a média da posição zero, por exemplo -- note que, como a média é dos três últimos valores, tentaríamos pegar as posições `0`, `-1` e `-2`, o que causaria uma `IndexOutOfBoundsException`.

Como essas informações são fundamentais para qualquer gráfico que plotemos, o gerador do modelo do gráfico receberá tais informações no construtor. Além delas, teremos também o objeto do `LineChartModel`, que guardará as informações do gráfico.

```
public class GeradorModeloGrafico {

  private SerieTemporal serie;
  private int comeco;
  private int fim;
  private LineChartModel modeloGrafico;

  public GeradorModeloGrafico(SerieTemporal serie, int comeco, int fim) {
    this.serie = serie;
    this.comeco = comeco;
    this.fim = fim;
    this.modeloGrafico = new LineChartModel();
  }

}
```

E, quem for usar essa classe, fará:

```
SerieTemporal serie = //...
GeradorModeloGrafico g = new GeradorModeloGrafico(serie, inicio, fim);
```

Repare como o código que usa o `GeradorModeloGrafico` não possui nada que o ligue ao `ChartModel` especificamente. O dia em que precisarmos mudar o gráfico a ser plotado ou mesmo mudar a tecnologia que gerará o gráfico, só precisaremos alterar a classe `GeradorModeloGrafico`. Relembre o conceito: esse é o poder do **encapsulamento**!

E nossa classe não se limitará a isso: ela encapsulará tudo o que for relacionado ao gráfico. Por exemplo, para criar o gráfico, precisamos ainda de um método que plote os pontos calculados por um indicador, como o `plotaMediaMovelSimples` abaixo. Ele passa por cada posição do `comeco` ao `fim` da `serie`, chamando o cálculo da média móvel simples.

```
public void plotaMediaMovelSimples() {
  MediaMovelSimples indicador = new MediaMovelSimples();
  LineChartSeries chartSerie = new LineChartSeries("MMS - Fechamento");

  for (int i = comeco; i <= fim; i++) {
    double valor = indicador.calcula(i, serie);
    chartSerie.set(i, valor);
  }
  this.modeloGrafico.addSeries(chartSerie);
  this.modeloGrafico.setLegendPosition("w");
  this.modeloGrafico.setTitle("Indicadores");
}
```

O método `plotaMediaMovelSimples` cria um objeto da `MediaMovelSimples` e varre a `SerieTemporal` recebida para calcular o conjunto de dados para o modelo do gráfico.

Por fim, ainda precisaremos de um método `getModeloGrafico` que devolverá o modelo para o `ArgentumBean`, já que o componente `p:chart` é que precisará desse objeto preenchido. Repare que o retorno é do tipo `ChartModel`, super classe do `LineChartModel`. É boa prática deixar nossa classe a mais genérica possível para funcionar com qualquer tipo de método.

> **Effective Java: refira-se a objetos pela sua interface**
>
> O item 34 do *Effective Java* discorre sobre preferir referenciar objetos pela sua interface, em vez de pelo seu próprio exato, sempre que este for relevante para o restante do sistema.
>
> O livro também menciona que, na falta de uma interface adequada, uma

superclasse pode ser utilizada, o que é nosso caso com a superclasse abstrata `ChartModel`. Referir-se a um objeto da forma mais genérica possível é uma boa ideia, já que isso faz com que futuras mudanças limitem-se ao máximo à classe que as encapsula.

Veja como fica o programa dentro do *ArgentumBean* e note que essa classe apenas delega para a outra toda a parte de criação do gráfico:

```
public class ArgentumBean {

  private List<Negociacao> negociacoes;
  private String indicadorBase;
  private String media;
  private ChartModel modeloGrafico;

  public ArgentumBean() {
    this.negociacoes = new ClienteWebService().getNegociacoes();
    List<Candle> candles = new CandleFactory().constroiCandles(negociacoes);
    SerieTemporal serie = new SerieTemporal(candles);

    GeradorModeloGrafico geradorGrafico =
          new GeradorModeloGrafico(serie, 2, serie.getUltimaPosicao());
    geradorGrafico.plotaMediaMovelSimples();
    this.modeloGrafico = geradorGrafico.getModeloGrafico();
  }

  public List<Negociacao> getNegociacoes() {
    return negociacoes;
  }

  public ChartModel getModeloGrafico() {
    return modeloGrafico;
  }
}
```

Note que, para quem usa o `GeradorModeloGrafico` nem dá para saber como ele gera o modelo. É um código **encapsulado**, **flexível**, **pouco acoplado** e **elegante**: usa boas práticas da Orientação a Objetos.

Faça sua pergunta.

9.6 - Para saber mais: Design Patterns Factory Method e Builder

Dois famosos design patterns do GoF são o **Factory Method** e o **Builder** -- e estamos usando ambos. Eles são chamados de **padrões de criação** (**creational patterns**), pois nos ajudam a criar objetos complicados.

A *factory* é usada pelo `GeradorDeSerie` dos testes. A ideia é que criar um objeto `SerieTemporal` diretamente é complicado. Então criaram um *método de fábrica* que encapsula essas complicações e já devolve o objeto prontinho para uso.

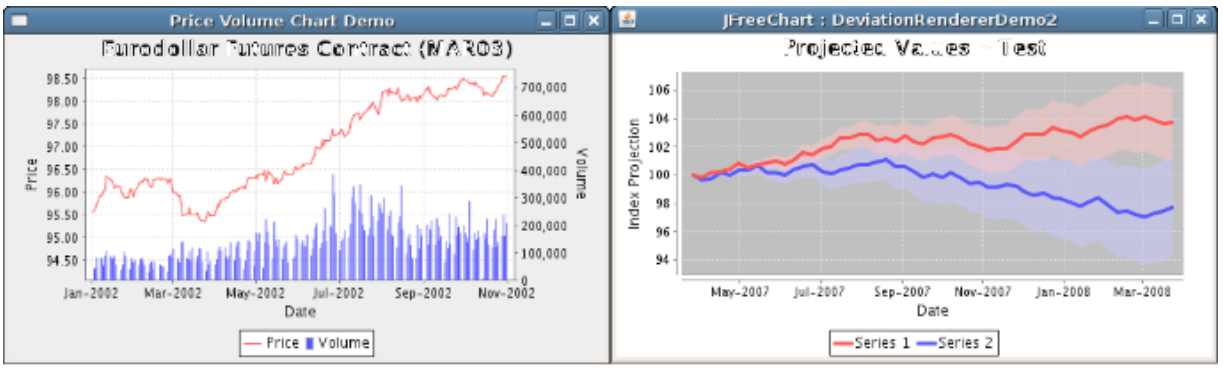
O padrão Builder é o que estamos usando na classe `GeradorModeloGrafico`. Queremos encapsular a criação complicada do modelo e que pode mudar depois com o tempo. Entra aí o *objeto construtor* da nossa classe Builder: seu único objetivo é descrever os passos para criação do nosso objeto final (o gráfico) e encapsular a complexidade disso.

Leia mais sobre esses e outros Design Patterns no livro do GoF.

JFreeChart

O **JFreeChart** é uma biblioteca famosa para desenho de gráficos, independente da tecnologia JSF. É um projeto de software livre iniciado em 2000 e que tem ampla aceitação pelo mercado, funcionando até em ambientes antigos que rodam Java 1.3.

Além do fato de ser livre, possui a vantagem de ser bastante robusta e flexível. É possível usá-la para desenhar gráficos de pontos, de barra, de torta, de linha, gráficos financeiros, gantt charts, em 2D ou 3D e muitos outros. Consegue dar saída em JPG, PNG, SVG, EPS e até mesmo exibir em componentes Swing.



O site oficial possui links para download, demos e documentação:

http://www.jfree.org/jfreechart/

Existe um livro oficial do JFreeChart escrito pelos desenvolvedores com exemplos e explicações detalhadas de vários gráficos diferentes. Ele é pago e pode ser obtido no site oficial. Além disso, há muitos tutoriais gratuitos na internet.

9.7 - Exercícios: Gráficos com Primefaces

1. Dentro da pasta `src/main/java` crie a classe **`GeradorModeloGrafico`** no pacote `br.com.caelum.argentum.grafico` para encapsular a criação do modelo do gráfico.

Use os recursos do Eclipse para escrever esse código! Abuse do `ctrl + espaço`, do `ctrl + 1` e do `ctrl + shift + O`.

```
public class GeradorModeloGrafico {
  // atributos: serie, comeco, fim e grafico
  // (todos gerados com ctrl + 1, conforme você criar o construtor)
  // importe as classes com ctrl + shift + O

  public GeradorModeloGrafico(SerieTemporal serie, int comeco, int fim) {
    this.serie = serie;
    this.comeco = comeco;
    this.fim = fim;
    this.modeloGrafico = new LineChartModel();
  }

  public void plotaMediaMovelSimples() {
    MediaMovelSimples indicador = new MediaMovelSimples();
    LineChartSeries chartSerie = new LineChartSeries("MMS - Fechamento");

    for (int i = comeco; i <= fim; i++) {
      double valor = indicador.calcula(i, serie);
      chartSerie.set(i, valor);
    }
    this.modeloGrafico.addSeries(chartSerie);
    this.modeloGrafico.setLegendPosition("w");
    this.modeloGrafico.setTitle("Indicadores");
  }

  public ChartModel getModeloGrafico() {
    return this.modeloGrafico;
  }
}
```

2. Agora que já temos uma classe que criará o modelo do gráfico, falta usá-la para

renderizar o gráfico na tela usando a tag do Primefaces. Abra a página `index.xhtml` que se encontra na pasta `WebContent`.

Na página procure o componente `h:body`. Logo depois da abertura da tag `h:body` e antes do `h:form` da lista de negociações, adicione o componente `p:chart`:

```
<p:chart type="line" model="#{argentumBean.modeloGrafico}" />
```

3. Na classe `ArgentumBean`, procure o construtor. É nele que criaremos uma `SerieTemporal` baseada na lista de negociações do Web Service. Depois usaremos o `GeradorModeloGrafico` para criar o modelo gráfico:

```
public ArgentumBean() {
  this.negociacoes = new ClienteWebService().getNegociacoes();
  List<Candle> candles = new CandleFactory().constroiCandles(negociacoes);
  SerieTemporal serie = new SerieTemporal(candles);

  GeradorModeloGrafico geradorGrafico =
        new GeradorModeloGrafico(serie, 2, serie.getUltimaPosicao());
  geradorGrafico.plotaMediaMovelSimples();
  this.modeloGrafico = geradorGrafico.getModeloGrafico();
}
```

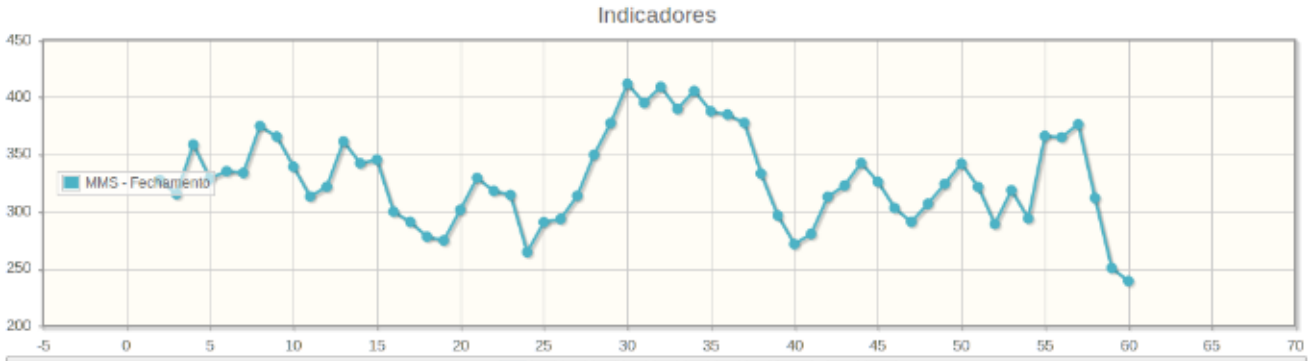
4. Você deve estar vendo um erro de compilação na última linha do seu construtor. Ele acontece porque faltou criarmos um novo atributo que representa o modelo do gráfico. Use o **ctrl + 1** para criar o atributo e, ao mesmo tempo, fazer o import da classe.

```
private ChartModel modeloGrafico;
```

Gere também o *getter* para este atributo, para que o componente `p:chart` consiga pegar o modelo do gráfico quando necessitar.

Para isso, na classe `ArgentumBean`, escreva `getM` e aperte **ctrl + espaço**. O Eclipse sugerirá o getter para o atributo `modeloGrafico`.

Salve o projeto. Reinicie o Tomcat e acesse em seu navegador: http://localhost:8080/fj22-argentum-web/index.xhtml



1 2 3 4 5 6 7 8 9 10

Preço	Quantidade	Volume	Data
246.93	14	3457.02	21/11/2013
376.8	22	8289.6	21/11/2013
411.78	24	9882.72	21/11/2013

CAPÍTULO ANTERIOR:

Refatoração: os Indicadores da bolsa

PRÓXIMO CAPÍTULO:

Aplicando Padrões de projeto

Facebook

Newsletter

Casa do Código

Twitter