05

Trabalhando com XML

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/laboratorio-java/stages/05/capitulo5.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Como consumir dados da bolsa ?

Como vamos puxar os dados da bolsa de valores para popular nossos *candles*?

Existem inúmeros formatos para se trabalhar com diversas bolsas. Sem dúvida, XML é um formato comumente encontrado em diversas indústrias, principalmente quando falamos da bolsa de valores.

Utilizaremos esse tal de XML. Para isso, precisamos conhecê-lo mais a fundo, seus objetivos, e como manipulá-lo. Considere que vamos consumir um arquivo XML como o que segue:

```
<list>
    <negociacao>
        <preco>43.5</preco>
        <quantidade>1000</quantidade>
        <data>
            <time>1222333777999</time>
        </data>
    </negociacao>
    <negociacao>
        <preco>44.1</preco>
        <quantidade>700</quantidade>
        <data>
            <time>1222444777999</time>
        </data>
    </negociacao>
    <negociacao>
        <preco>42.3</preco>
        <quantidade>1200</quantidade>
        <data>
            <time>1222333999777</time>
        </data>
    </negociacao>
</list>
```

Uma lista de negociações! Cada negociação informa o preço, quantidade e uma data. Essa data é composta por um horário dado no formato de `Timestamp`, e opcionalmente um `Timezone`.

O formato XML

XML (eXtensible Markup Language) é uma formalização da W3C para gerar linguagens de marcação que podem se adaptar a quase qualquer tipo de necessidade. Algo bem extensível, flexível, de fácil leitura e hierarquização. Sua definição formal pode ser encontrada em:

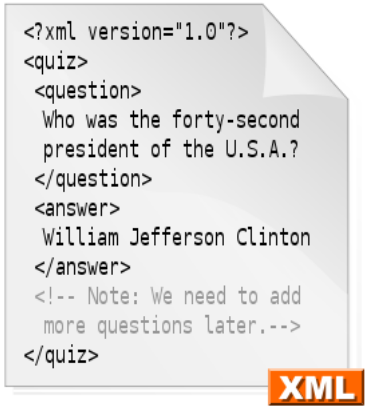
http://www.w3.org/XML/ (http://www.w3.org/XML/)

Exemplo de dados que são armazenados em XMLs e que não conhecemos tão bem, é o formato aberto de gráficos vetoriais, o **SVG** (usado pelo Corel Draw, Firefox, Inkscape, etc), e o Open Document Format (ODF), formato usado pelo OpenOffice, e hoje em dia um padrão ISO de extrema importância (na verdade o ODF é um ZIP que contém XMLs internamente).

A ideia era criar uma linguagem de marcação que fosse muito fácil de ser lida e gerada por softwares, e pudesse ser integrada às outras linguagens. Entre seus princípios básicos, definidos pelo W3C:

- Separação do conteúdo da formatação;
- Simplicidade e legibilidade;
- Possibilidade de criação de tags novas;
- Criação de arquivos para validação (DTDs e schemas).

O XML é uma excelente opção para documentos que precisam ter seus dados organizados com uma certa hierarquia (uma árvore), com relação de pai-filho entre seus elementos. Esse tipo de arquivo é dificilmente organizado com CSVs ou *properties*. Como a própria imagem do Wikipedia nos traz e mostra o uso estruturado e encadeado de maneira hierárquica do XML:



O cabeçalho opcional de todo XML é o que se segue:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Esses caracteres não devem ser usados como elemento, e devem ser "escapados":

```
&, use &amp;
', use &apos;
", use &quot;
<, use &lt;
>, use &gt;
```

Você pode, em Java, utilizar a classe `String` e as regex do pacote `java.util.regex` para criar um programa que lê um arquivo XML. Isso é uma grande perda de tempo, visto que o Java, assim como quase toda e qualquer linguagem existente, possui uma ou mais formas de ler um XML. O Java possui diversas, mas aqui vamos usar uma das mais famosas dela, o **XStream**.

Lendo XML de modo fácil: XStream

O **XStream** é uma alternativa perfeita para os casos de uso de XML em persistência, transmissão de dados e configuração. Sua facilidade de uso e performance elevada são os seus principais atrativos.

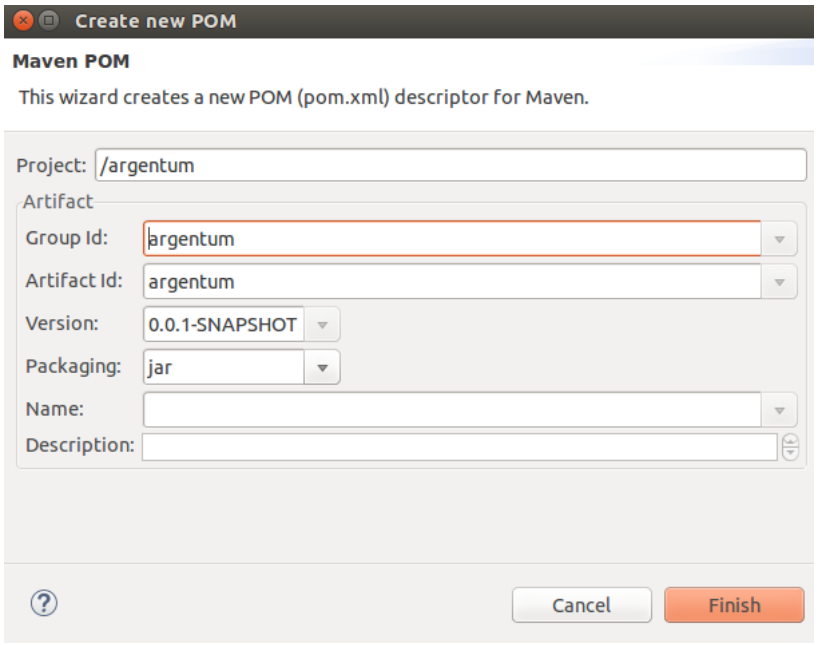
Diversos projetos *open source*, como o *container* de inversão de controle **NanoContainer**, o *framework* de redes neurais **Joone**, dentre outros, passaram a usar **XStream** depois de experiências com outras bibliotecas. O **XStream** é conhecido pela sua extrema facilidade de uso. Repare que raramente precisaremos fazer configurações ou mapeamentos, como é extremamente comum nas outras bibliotecas mesmo para os casos mais básicos.

Configurando nosso projeto como um Maven Project

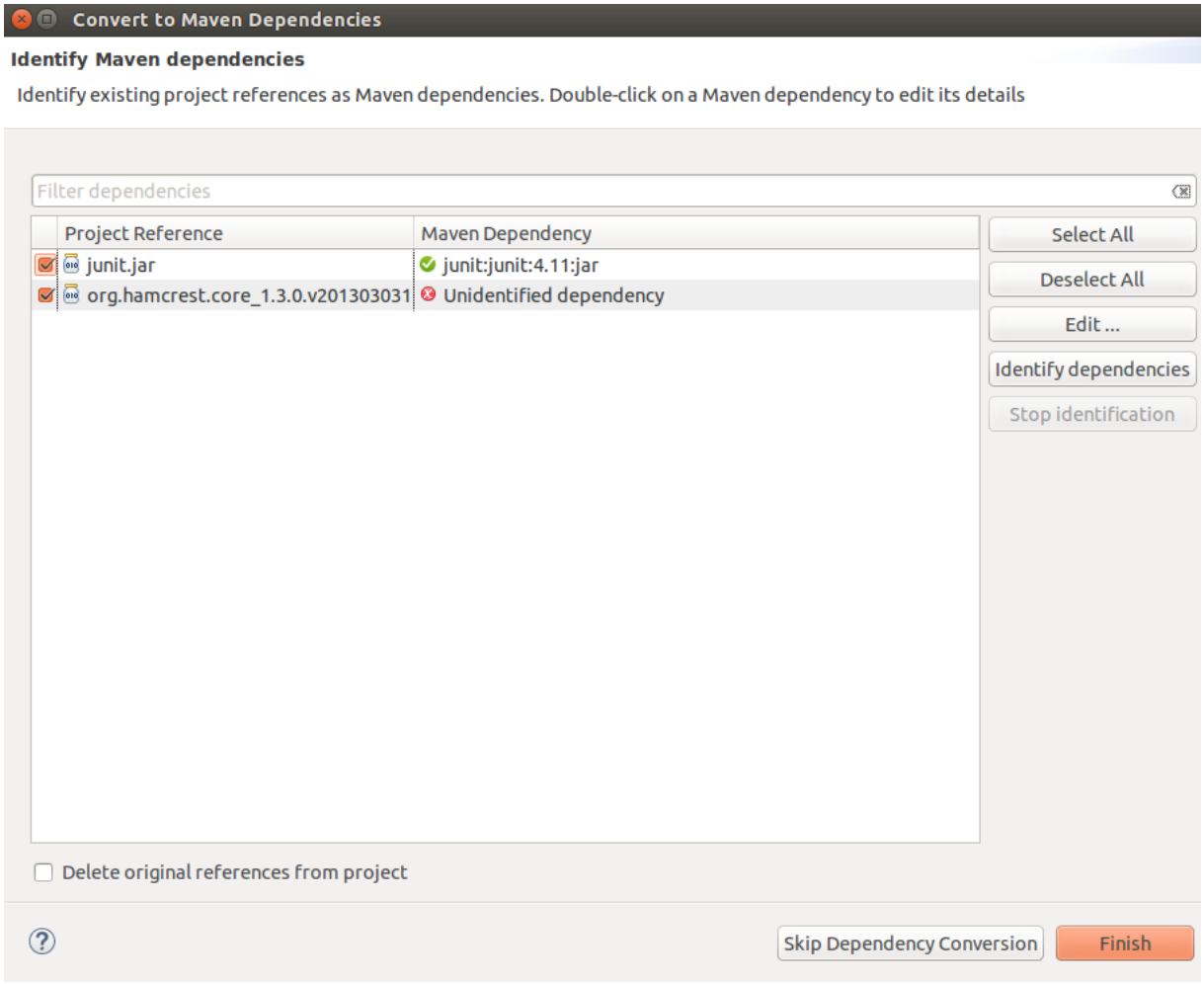
Então de início, vamos aprender a configurar o nosso projeto para utilizar o XStream. Primeiro, vamos alterar a configuração do nosso projeto para que ele se torne um **Maven Project**. O Maven, caso você não conheça, é um famoso gerenciador de projetos do mundo Java, capaz de gerenciar dependências, o processo de *build* e até mesmo a documentação do seu projeto. Você pode aprender mais sobre o Maven [neste curso do Alura (https://cursos.alura.com.br/course/maven)](https://cursos.alura.com.br/course/maven).

Para trocar o nosso projeto para um **Maven Project**, clique com o botão direito em cima do projeto e depois em *Configure* > *Convert to Maven Project* .

Na próxima tela clique em *Finish*:



E em seguida, clique novamente em *Finish* e espere ele baixar as dependências:



Agora que o nosso projeto já é um **Maven Project**, podemos alterar o `pom.xml` para incluir novas dependências, no nosso caso queremos adicionar a biblioteca **XStream**. Adicione o código abaixo no seu `pom.xml`, dentro de `<dependencies>`. Caso o seu pom não tenha sido criado com o nó dependencies, basta adiciona-lo manualmente e colocar o trecho de código abaixo dentro dele :

```
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.4.9</version>
</dependency>
```

Depois de adicionada a dependência no `pom.xml`, clique com o botão direito em cima do seu projeto e vá no menu *Maven* > *Update Project...* e clique em **OK**. Agora o XStream já está disponível para usarmos no nosso projeto.

Utilizando o XStream

Como gerar o XML de uma negociação? Primeiramente devemos ter uma referência para o objeto. Podemos simplesmente criá-lo e populá-lo ou então deixar que o Hibernate faça isso.

Com a referência `negociacao` em mãos, basta agora pedirmos ao XStream que gere o XML correspondente:

```
Negociacao negociacao = new Negociacao(40.5, 100, LocalDateTime.now());

XStream stream = new XStream(new DomDriver());
System.out.println(stream.toXML(negociacao));
```

E obtemos como resposta algo parecido com:

```
<br.com.caelum.argentum.modelo.Negociacao>
    <preco>40.5</preco>
    <quantidade>100</quantidade>
    <data resolves-to="java.time.Ser">
        <byte>5</byte>
        <int>2016</int>
        <byte>4</byte>
        <byte>6</byte>
        <byte>16</byte>
        <byte>33</byte>
        <byte>10</byte>
        <int>91000000</int>
    </data>
</br.com.caelum.argentum.modelo.Negociacao>
```

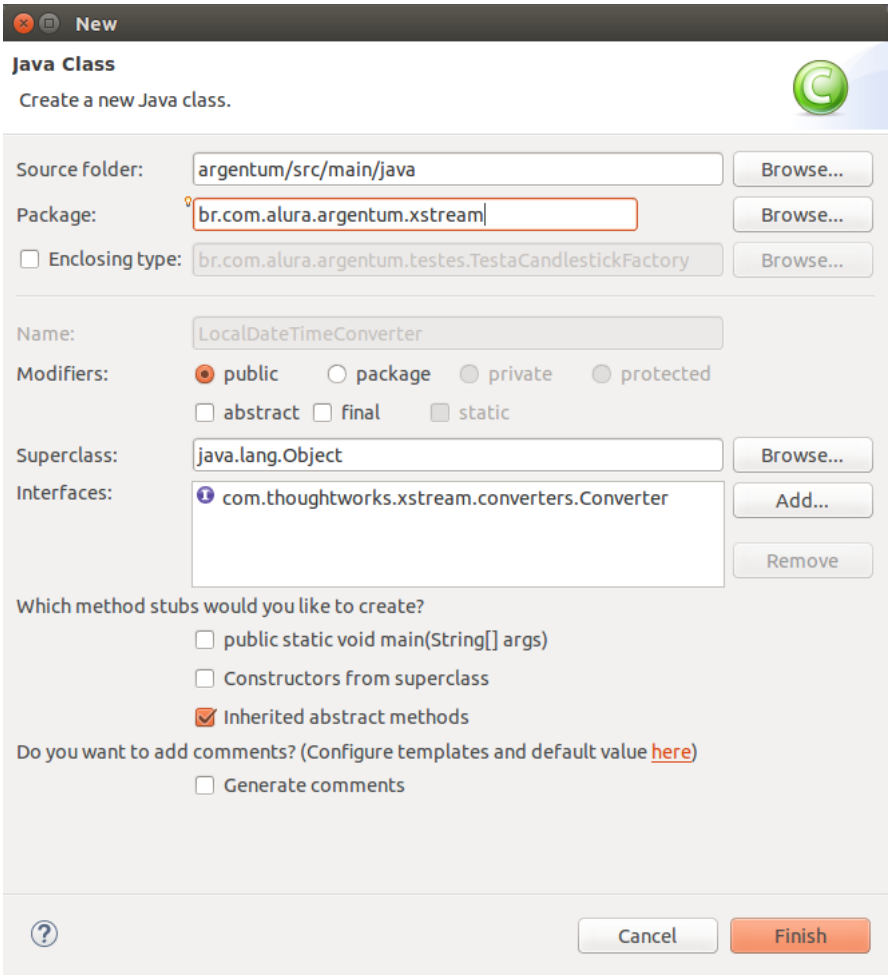
Vemos que o `preço` e a `quantidade` estão corretos no XML, porém a data não está como esperávamos. Isto acontece porque o XStream ainda não sabe como interpretar a nova API de data do Java 8 , o `LocalDateTime` que estamos utilizando. Para solucionar esse problema, iremos criar um **Converter**, que será responsável por *traduzir* um `LocalDateTime` para o formato que esperamos.

Criando um conversor do XStream

Utilizando um método do próprio XStream para registar um conversor local, vamos dizer que queremos utilizar o campo `data` , da classe `Negociacao` , e vamos chamar nosso *converter* de `LocalDateTimeConverter` :

```
Negociacao negociacao = new Negociacao(40.5, 100, LocalDateTime.now());

XStream stream = new XStream(new DomDriver());
stream.registerLocalConverter(Negociacao.class, "data",
        new LocalDateTimeConverter());

System.out.println(stream.toXML(negociacao));
```

Lembre-se de utilizar os atalhos do Eclipse, como o **CTRL + Espaço** para completar o nome do método e quando ele reclamar que a classe `LocalDateTimeConverter` não existe, utilize o **CTRL + 1** para criar a classe que iremos implementar. Crie no pacote `br.com.alura.argentum.xstream` :



Você verá que automaticamente o Eclipse irá preencher a classe com 3 métodos que precisamos implementar, o `canConvert`, `marshal` e o `unmarshal`.

Vamos implementar um por vez. O `canConvert` é o método que verifica se o dado que estamos convertendo é da classe correta, então vamos implementá-lo de um modo que só aceitemos `LocalDateTime` para o campo `data`:

```
@Override
public boolean canConvert(Class type) {
    return type.isAssignableFrom(LocalDateTime.class);
}
```

Então se a classe for do tipo `LocalDateTime`, o método retornará **true** e poderemos começar a conversão.

Agora os métodos `marshal` e `unmarshal`, que são responsáveis por serializar e desserializar o nosso objeto, respectivamente. Como queremos transformar um objeto do nosso modelo em um XML, o método `marshal` recebe 3 parâmetros: o objeto que queremos transformar (`LocalDateTime`), um *writer*, que é o responsável por escrever no nosso XML e um contexto, que não vamos precisar agora.

Como quereremos um XML com essa cara:

```
<negociacao>
    <preco>43.5</preco>
    <quantidade>1000</quantidade>
    <data>
        <time>1222333777999</time>
        <timezone>Etc/UTC</timezone>
```

```
    </data>
</negociacao>
```

Precisamos adaptar primeiro a nossa data, para ser exibida em *UNIX Timestamp*, que é o número em segundos desde 1 de Janeiro de 1970, que é um padrão bastante utilizado para marcar tempo na computação. Como estamos simulando um sistema preciso como o da bolsa de valores, o nosso XML incluí até os **milissegundos** , o que devemos tomar cuidado, pois a classe `LocalDateTime` só trabalha com segundos! Além disso, também precisamos do *timezone* ou em português claro, o fuso horário, que também não vem com a classe `LocalDateTime` . Para cuidar disso, vamos utilizar uma outra classe, que possui fácil acesso tanto aos milissegundos quanto aos fusos horários: a `ZonedDateTime` .

Começando a implementação do método `marshall` :

```
@Override
public void marshal(Object object, HierarchicalStreamWriter writer,
        MarshallingContext context) {

    LocalDateTime data = (LocalDateTime) object;
    ZonedDateTime dataComZona = data.atZone(ZoneOffset.systemDefault());
    long millis = dataComZona.toInstant().toEpochMilli();
}
```

Fazemos um casting do nosso Objeto recebido para um `LocalDateTime` , transformando-o em um `ZonedDateTime` em seguida, a partir do fuso horário do sistema. Depois utilizamos os métodos que a classe nos fornece para obter a quantidade de **milissegundos** desde 1 de Janeiro de 1970 e salvamos na variável `millis` .

Como queremos montar o XML com 2 nós dentro de data, vamos utilizar o `writer` para escrever o nó `<time>` e depois escrever o nó `<timezone>` logo abaixo:

```
@Override
public void marshal(Object object, HierarchicalStreamWriter writer,
        MarshallingContext context) {

    LocalDateTime data = (LocalDateTime) object;
    ZonedDateTime dataComZona = data.atZone(ZoneOffset.systemDefault());
    long millis = dataComZona.toInstant().toEpochMilli();

    writer.startNode("time");
    writer.setValue(String.valueOf(millis));
    writer.endNode();
}
```

E na sequencia, vamos montar o novo nó `<timezone>` :

```
@Override
public void marshal(Object object, HierarchicalStreamWriter writer,
        MarshallingContext context) {

    LocalDateTime data = (LocalDateTime) object;
    ZonedDateTime dataComZona = data.atZone(ZoneOffset.systemDefault());
    long millis = dataComZona.toInstant().toEpochMilli();
```

```
        writer.startNode("time");
        writer.setValue(String.valueOf(millis));
        writer.endNode();

        writer.startNode("timezone");
        writer.setValue(dataComZona.getZone().toString());
        writer.endNode();
    }
```

Com este método pronto, já podemos montar um pequeno teste para verificar se nossa correção está funcionando:

```
public class LocalDateTimeConverterTest {

    @Test
    public void deveGerarOXmlComADataAdequada() {

    }
}
```

Vamos criar uma negociação e o xml que esperamos que ela se transforme:

```
public class LocalDateTimeConverterTest {

    @Test
    public void deveGerarOXmlComADataAdequada() {

        String resultadoExperado =  "<negociacao>\n" +
                                    "  <preco>40.5</preco>\n" +
                                    "  <quantidade>100</quantidade>\n" +
                                    "  <data>\n" +
                                    "    <time>1459479600000</time>\n" +
                                    "    <timezone>America/Sao_Paulo</timezone>\n" +
                                    "  </data>\n" +
                                    "</negociacao>";

        Negociacao negociacao = new Negociacao(40.5, 100, LocalDateTime.of(
                2016, 04, 01, 00, 00));
    }
}
```

E vamos utilizar o XStream para *converter* para XML, sem esquecer de habilitar o nosso *converter*:

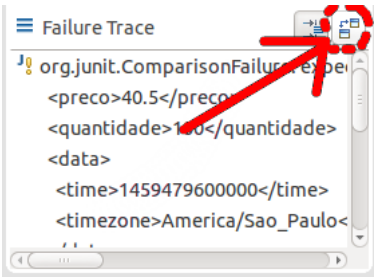
```
public class LocalDateTimeConverterTest {

    @Test
    public void deveGerarOXmlComADataAdequada() {

        String resultadoExperado =  "<negociacao>\n" +
                                    "  <preco>40.5</preco>\n" +
                                    "  <quantidade>100</quantidade>\n" +
                                    "  <data>\n" +
                                    "    <time>1459479600000</time>\n" +
                                    "    <timezone>America/Sao_Paulo</timezone>\n" +
```

```
                "  </data>\n" +
                "</negociacao>";

        Negociacao negociacao = new Negociacao(40.5, 100, LocalDateTime.of(
                2016, 04, 01, 00, 00));

        XStream stream = new XStream(new DomDriver());
        stream.registerLocalConverter(Negociacao.class, "data",
                new LocalDateTimeConverter());
        String resultado = stream.toXML(negociacao);
    }
}
```

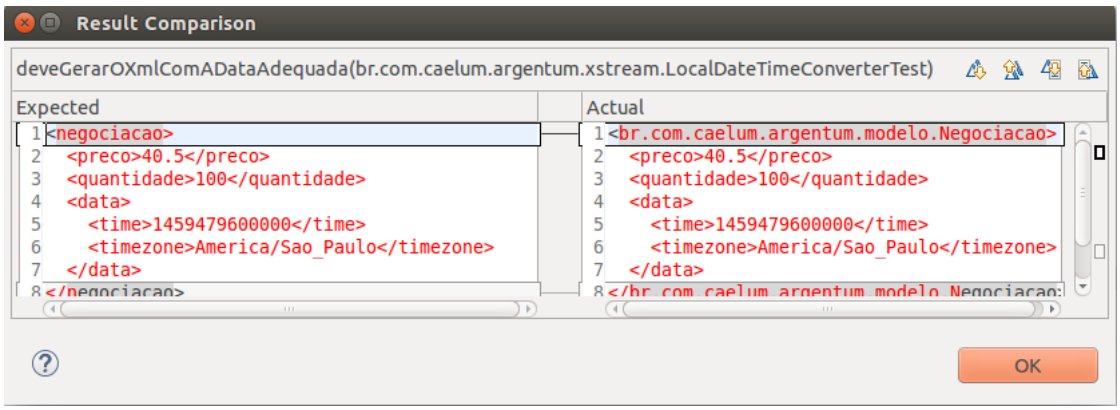
E agora com o método `assertEquals` , comparamos o resultado esperado com o resultado obtido:

```
public class LocalDateTimeConverterTest {

    @Test
    public void deveGerarOXmlComADataAdequada() {

        String resultadoExperado =  "<negociacao>\n" +
                                    "  <preco>40.5</preco>\n" +
                                    "  <quantidade>100</quantidade>\n" +
                                    "  <data>\n" +
                                    "    <time>1459479600000</time>\n" +
                                    "    <timezone>America/Sao_Paulo</timezone>\n" +
                                    "  </data>\n" +
                                    "</negociacao>";

        Negociacao negociacao = new Negociacao(40.5, 100, LocalDateTime.of(
                2016, 04, 01, 00, 00));

        XStream stream = new XStream(new DomDriver());
        stream.registerLocalConverter(Negociacao.class, "data",
                new LocalDateTimeConverter());
        String resultado = stream.toXML(negociacao);

        assertEquals(resultadoExperado, resultado);
    }
}
```

Rodamos o teste e vemos que ele... falha! Vamos clicar no botão abaixo para comparar o resultado:



Ele abrirá a seguinte janela, na qual podemos comparar os resultados:



Aqui vemos que a única coisa de diferente é por que o XStream cria a tag com nome completo da classe , em vez de apenas `negociacao` . Mas podemos facilmente configurar isso definindo um `alias` , dizendo o nome que queremos para a tag , com base na classe que ele espera:

```
@Test
public void deveGerarOXmlComADataAdequada() {

    // restante do código

    XStream stream = new XStream(new DomDriver());
    stream.alias("negociacao", Negociacao.class);
    stream.registerLocalConverter(Negociacao.class, "data",
            new LocalDateTimeConverter());

    String resultado = stream.toXML(negociacao);

    Assert.assertEquals(resultadoExperado, resultado);
}
```

Testando novamente... O teste passa :)

Agora falta implementarmos o método oposto, o `unmarshal` , que pega uma string de texto do XML e transforma em um objeto, no nosso caso um `LocalDateTime` .

Vamos começar lendo o XML:

```
@Override
public Object unmarshal(HierarchicalStreamReader reader,
        UnmarshallingContext context) {

    reader.moveDown();
    String time = reader.getValue();
    reader.moveUp();

    reader.moveDown();
    String timezone = reader.getValue();
    reader.moveUp();

    return  null;
}
```

Descemos para cada um dos nós, lemos seu conteúdo e subimos novamente. Agora com os dados em mãos, basta fazer as conversões opostas do método `marshal`, transformando uma `String` em `long`, para em seguida criar um `ZoneDateTime` a partir de um `Instant` e uma `Timezone`.

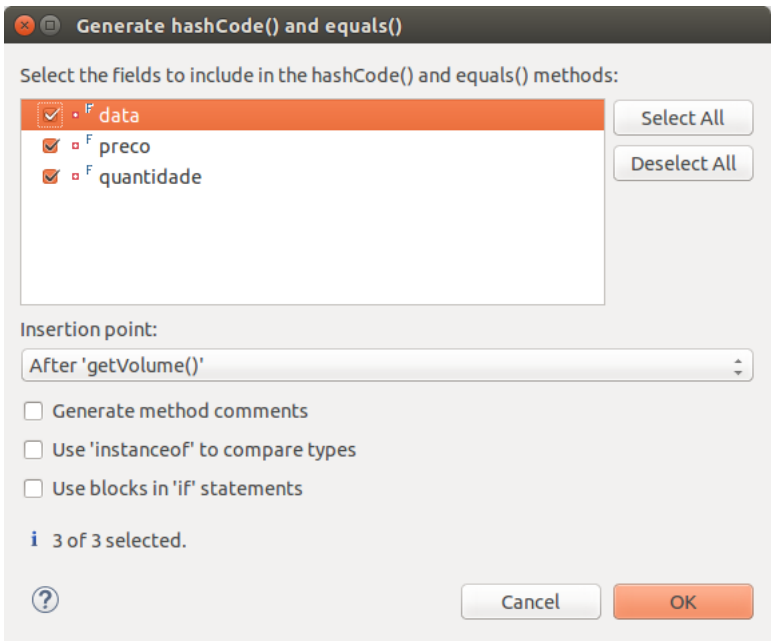
Daí, é só converter de volta de `ZoneDateTime` para `LocalDateTime` e retonar o objeto!

```
@Override
public Object unmarshal(HierarchicalStreamReader reader,
        UnmarshallingContext context) {

    reader.moveDown();
    String time = reader.getValue();
    reader.moveUp();

    reader.moveDown();
    String timezone = reader.getValue();
    reader.moveUp();

    long tempoEmMillis = Long.parseLong(time);
    Instant instante = Instant.ofEpochMilli(tempoEmMillis);
    ZoneId zona = ZoneId.of(timezone);
    ZonedDateTime dataComZone = ZonedDateTime.ofInstant(instante,zona);
    LocalDateTime data = dataComZone.toLocalDateTime();

    return  data;
}
```

Montando um teste `deveGerarObjetoAdequadoDeUmXml` para verificar se tudo está correto:

```
@Test
public void deveGerarObjetoAdequadoDeUmXml() {

    String xml = "<negociacao>\n" +
                 "  <preco>40.5</preco>\n" +
                 "  <quantidade>100</quantidade>\n" +
                 "  <data>\n" +
                 "    <time>1459479600000</time>\n" +
                 "    <timezone>America/Sao_Paulo</timezone>\n" +
                 "  </data>\n" +
                 "</negociacao>";

    XStream stream = new XStream(new DomDriver());
    stream.alias("negociacao", Negociacao.class);
    stream.registerLocalConverter(Negociacao.class, "data",
            new LocalDateTimeConverter());
    Negociacao negociacaoGerada = (Negociacao) stream.fromXML(xml);

    Negociacao negociacaoEsperada = new Negociacao(40.5, 100,
            LocalDateTime.of(2016, 04, 01, 00, 00));

    Assert.assertEquals(negociacaoEsperada, negociacaoGerada);
}
```

Utilizamos o mesmo XML e a `negociacao` anteriores, só que desta vez invertemos o processo. Aqui, como estamos comparando duas negociações, precisamos reescrever o método `equals`, para que o `assertEquals` saiba como comparar estes dois objetos. Com o Eclipse, isto é fácil, basta ir na classe `Negociacao` e utilizar o atalho **CTRL + 3** e começar a escrever *equals...* e ele irá sugerir a opção *Generate hashCode() and equals()...*, que você deve selecionar e deixar todos os campos marcados. Ele gerará os métodos automaticamente:



Agora podemos comparar objetos `Negociacao` nos nossos testes. Rodamo-oss e tudo ocorre como esperado! Estamos serializando e desserializando as negociações corretamente.

Criando um leitor para várias negociações

Vamos criar uma classe responsável por fazer a leitura de várias negociações, ela será o `LeitorXML`. Ela encapsulará a leitura de um XML, e retornará uma lista de negociações. Crie a classe `LeitorXML` no pacote `br.com.alura.argentum.reader` e comece a implementação do método `carrega`:

```
public class LeitorXML {

    public List<Negociacao> carrega(InputStream inputStream) {

        XStream stream = new XStream(new DomDriver());
        stream.alias("negociacao", Negociacao.class);
        stream.registerLocalConverter(Negociacao.class, "data",
                new LocalDateTimeConverter());

        return (List<Negociacao>) stream.fromXML(inputStream);
    }
}
```

Não esqueça de registrar o *converter*!

A classe `InputStream` é uma classe abstrata, que deve ser implementada por quem for utilizar o leitor de XML. A sua grande vantagem é que o utilizador pode escolher se quer ler de uma fonte de dados de um `File`, ou de *buffer* de dados da internet, ou uma `String` - como que veremos no teste abaixo - deixando a implementação mais específica a carga do desenvolvedor que for utilizá-la.

Implementando um teste para o nosso `LeitorXML`:

```
@Test
public void carregaXMLComUmaNegociacaoApenas() {

    String xmlDeTeste = "<list>\n" +
                        "  <negociacao>\n" +
                        "    <preco>40.5</preco>\n" +
                        "    <quantidade>100</quantidade>\n" +
                        "    <data>\n" +
                        "      <time>1459479600000</time>\n" +
                        "      <timezone>America/Sao_Paulo</timezone>\n" +
                        "    </data>\n" +
                        "  </negociacao>\n" +
                        "</list>";

    Negociacao negociacaoEsperada = new Negociacao(40.5, 100,
            LocalDateTime.of(2016, 04, 01, 00, 00));

    LeitorXML leitor = new LeitorXML();

    InputStream xml = new ByteArrayInputStream(xmlDeTeste.getBytes());

    List<Negociacao> negociacoes = leitor.carrega(xml);

    assertEquals(1, negociacoes.size());
    assertEquals(negociacaoEsperada, negociacoes.get(0));
}
```

Aqui pedimos ao nosso leitor que interprete um XML de uma lista de negociações com apenas uma negociação. Depois de montar a lista, fazemos o *assert* comparando o tamanho da lista e se recebemos a negociação esperada.

Verificamos e o teste passa! Parece que nosso leitor funciona corretamente.

Neste capítulo aprendemos um pouco sobre como funciona a biblioteca XStream, mas se você quer se especializar nela, ou se aprofundar em algum assunto específico - como *Converters* - você pode assistir ao curso do Alura focado em XStream (https://cursos.alura.com.br/course/xstream).

O que aprendemos?

- Como iremos consumir os dados da bolsa de valores.
- O formato XML.
- A biblioteca XStream.
- Como criar um *converter* customizado do XStream.
- Como fazer testes para os seus XMLs.
- Mais classes do pacote `java.time`.
- Sobrescrevendo o método `equals()`.