



Trabalhando com XML

*"Se eu enxerguei longe, foi por ter subido nos ombros de gigantes."
-- Isaac Newton*

1.1 - Os dados da bolsa de valores

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, inclusive na 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>
  <negocio>
    <preco>43.5</preco>
    <quantidade>1000</quantidade>
    <data>
      <time>555454646</time>
    </data>
  </negocio>
  <negocio>
    <preco>44.1</preco>
    <quantidade>700</quantidade>
    <data>
      <time>555454646</time>
    </data>
  </negocio>
  <negocio>
    <preco>42.3</preco>
    <quantidade>1200</quantidade>
    <data>
```

```

                                <time>559329406</time>
                                </data>
                                </negocio>
</list>
```

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

1.2 - 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:

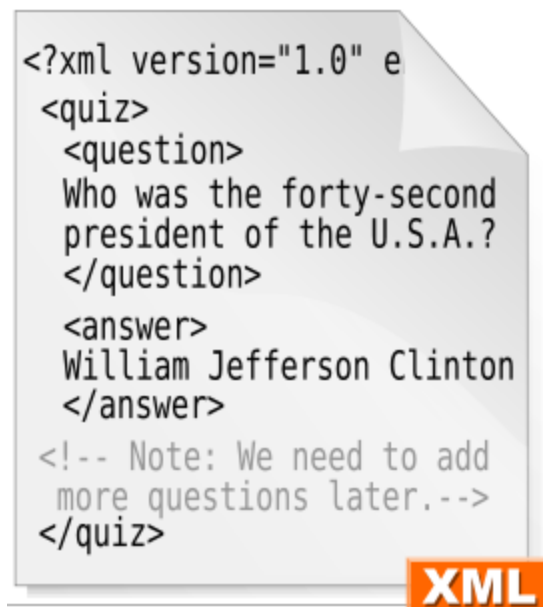
<http://www.w3.org/XML/>

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 as 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 trás 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 &
- ', use '
- ", use "
- <, use <
- >, use >

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, vamos ver algumas delas, suas vantagens e suas desvantagens.

1.3 - Lendo XML com Java de maneira difícil, o SAX

O SAX (**Simple API for XML**) é uma API para ler dados em XML, também conhecido como **Parser de XML**. Um parser serve para analisar uma estrutura de dados e geralmente o que fazemos é transformá-la em uma outra.

Neste processo de análise também podemos ler o arquivo XML para procurar algum determinado elemento e manipular seu conteúdo.

O parser lê os dados XML como um fluxo ou uma sequência de dados. Baseado no conteúdo lido, o parser vai disparando eventos. É o mesmo que dizer que o parser SAX funciona orientado a eventos.

Existem vários tipos de eventos, por exemplo:

- início do documento XML;
- início de um novo elemento;
- novo atributo;
- início do conteúdo dentro de um elemento.

Para tratar estes eventos, o programador deve passar um objeto **listener** ao parser que será notificado automaticamente pelo parser quando um desses eventos ocorrer. Comumente, este objeto é chamado de **Handler**, **Observer**, ou **Listener** e é quem faz o trabalho necessário de processamento do XML.

```
public class NegociacaoHandler extends DefaultHandler {  
  
    @Override  
    public void startDocument() throws SAXException {  
  
    }  
  
    @Override  
    public void startElement(String uri, String localName,  
                             String name, Attributes attributes) throws SAXException {  
        // aqui voce é avisado, por exemplo  
        // do inicio da tag "<preco>"  
    }  
}
```

```

@Override
public void characters(char[] chars, int offset, int len) throw
    // aqui voce seria avisado do inicio
    // do conteudo que fica entre as tags, como por exemplo
    // de dentro de "<preco>30</preco>"

    // para saber o que fazer com esses dados, voce precisa
    // guardado em algum atributo qual era a negociação que
    // sendo percorrida
}

@Override
public void endElement(String uri, String localName, String name)
    // aviso de fechamento de tag
}
}

```

A classe `DefaultHandler` permite que você reescreva métodos que vão te notificar sobre quando um elemento (tag) está sendo aberto, quando está sendo fechado, quando caracteres estão sendo parseados (conteúdo de uma tag), etc.. Você é o responsável por saber em que posição do object model (árvore) está, e que atitude deve ser tomada. A interface `ContentHandler` define mais alguns outros métodos.

Curiosidade sobre o SAX

Originalmente o SAX foi escrito só para Java e vem de um projeto da comunidade (<http://www.saxproject.org>), mas existem outras implementações em C++, Perl e Python.

O SAX está atualmente na versão 2 e faz parte do JAX-P (Java API for XML Processing).

O SAX somente sabe ler dados e nunca modificá-los e só consegue ler para frente, nunca para trás. Quando passou um determinado pedaço do XML que já foi lido, não há mais como voltar. O parser SAX não guarda a estrutura do documento XML na memória.

Também não há como fazer acesso aleatório aos itens do documento XML, somente sequencial.

Por outro lado, como os dados vão sendo analisados e transformados (pelo Handler) na hora da leitura, o SAX ocupa pouca memória, principalmente porque nunca vai conhecer o documento inteiro e sim somente um pequeno pedaço. Devido também a leitura sequencial, ele

é muito rápido comparado com os parsers que analisam a árvore do documento XML completo.

Quando for necessário ler um documento em partes ou só determinado pedaço e apenas uma vez, o SAX parser é uma excelente opção.

StAX - Streaming API for XML

StAX é um projeto que foi desenvolvido pela empresa BEA e padronizado pela JSR-173. Ele é mais novo do que o SAX e foi criado para facilitar o trabalho com XML. StAX faz parte do Java SE 6 e JAX-P.

Como o SAX, o StAX também lê os dados de maneira sequencial. A diferença entre os dois é a forma como é notificada a ocorrência de um evento.

No Sax temos que registrar um `Handler`. É o SAX que avisa o `Handler` e chama os métodos dele. Ele empurra os dados para o `Handler` e por isso ele é um parser do tipo `PUSH`, .

O StAX, ao contrário, não precisa deste `Handler`. Podemos usar a API do StAX para chamar seus métodos, quando e onde é preciso. O cliente decide, e não o parser. É ele quem pega/tira os dados do StAX e por isso é um parser do tipo `PULL`.

O site <http://www.xmlpull.org> fornece mais informações sobre a técnica de **Pull Parsing**, que tem sido considerada por muitos como a forma mais eficiente de processar documentos xml.

A biblioteca XPP3 é a implementação em Java mais conhecida deste conceito. É usada por outras bibliotecas de processamento de xml, como o CodeHaus XStream.

1.4 - 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.

É um projeto hospedado na Codehaus, um repositório de código open source focado em Java, que foi formado por desenvolvedores de famosos projetos como o XDoclet, PicoContainer e Maven. O grupo é patrocinado por empresas como a ThoughtWorks, BEA e Atlassian. Entre os seis desenvolvedores do projeto, Guilherme Silveira da Caelum está também presente.

<http://xstream.codehaus.org>

Diversos projetos opensource, 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.

Como gerar o XML de uma negociação? Primeiramente devemos ter uma referência ao bean. Podemos simplesmente criar um e populá-lo, ou este pode ser, por exemplo, uma entidade do Hibernate.

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

```
Negocio negocio = new Negocio(42.3, 100, Calendar.getInstance());

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

E o resultado é:

```
<br.com.caelum.argentum.Negocio>
  <preco>42.3</preco>
  <quantidade>100</quantidade>
  <data>
    <time>1220009639873</time>
    <timezone>America/Sao_Paulo</timezone>
  </data>
</br.com.caelum.argentum.Negocio>
```

A classe XStream atua como **façade** de acesso para os principais recursos da biblioteca. O construtor da classe XStream recebe como argumento um `Driver`, que é a engine que vai gerar/consumir o XML. Aqui você pode definir se quer usar SAX, DOM, DOM4J dentre outros, e com isso

o XStream vai ser mais rápido, mais lento, usar mais ou menos memória, etc.

O default do XStream é usar um driver chamado XPP3, desenvolvido na universidade de Indiana e conhecido por ser extremamente rápido (leia mais no box de pull parsers). Para usá-lo você precisa de um outro JAR no classpath do seu projeto.

O método `toXML` retorna uma `String`. Isso pode gastar muita memória no caso de você serializar uma lista grande de objetos. Ainda existe um overload do `toXML`, que além de um `Object` recebe um `OutputStream` como argumento para você poder gravar diretamente num arquivo, socket, etc.

Diferentemente de outros parsers do Java, o XStream serializa por default os objetos através de seus atributos (sejam privados ou não), e não através de getters e setters.

Repare que o XStream gerou a tag raiz com o nome de `br.com.caelum.argentum.Negocio`. Isso porque não existe um conversor para ela, então ele usa o próprio nome da classe e gera o XML recursivamente para cada atributo não transiente daquela classe.

Porém, muitas vezes temos um esquema de XML já muito bem definido, ou simplesmente não queremos gerar um XML com cara de java. Para isso podemos utilizar um `alias`. Vamos modificar nosso código que gera o XML:

```
XStream stream = new XStream(new DomDriver());
stream.alias("negocio", Negocio.class);
```

Essa configuração também pode ser feita através da anotação `@XStreamAlias("negocio")` em cima da classe `Negocio`.

Podemos agora fazer o processo inverso. Dado um XML que representa um bean da nossa classe `Negocio`, queremos popular esse bean. O código é novamente extremamente simples:

```
XStream stream = new XStream(new DomDriver());
stream.alias("negocio", Negocio.class);
Negocio n = (Negocio) stream.fromXML("<negocio>" +
                                     "<preco>42.3</preco>" +
                                     "<quantidade>100</quantidade>");
```



```
                                "</negocio>");  
System.out.println(negocio.getPreco());
```

Obviamente não teremos um XML dentro de um código Java. O exemplo aqui é meramente ilustrativo (útil em um teste!). Os atributos não existentes ficarão como `null` no objeto, como é o caso aqui do atributo `data`, ausente no XML.

O XStream possui um overload do método `fromXML` que recebe um `InputStream` como argumento, outro que recebe um `Reader`.

JAXB ou XStream?

A vantagem do JAXB é ser uma especificação da Sun, e a do XStream é ser mais flexível e deixar trabalhar com classes imutáveis.

@XstreamAlias

Ao invés de chamar `stream.alias("negocio", Negocio.class);`, podemos fazer essa configuração direto na classe `Negocio` com uma anotação:

```
@XstreamAlias("negocio")  
public class Negocio {  
}
```

Para habilitar o suporte a anotações, precisamos chamar no `xstream`:

```
stream.autodetectAnnotations(true);
```

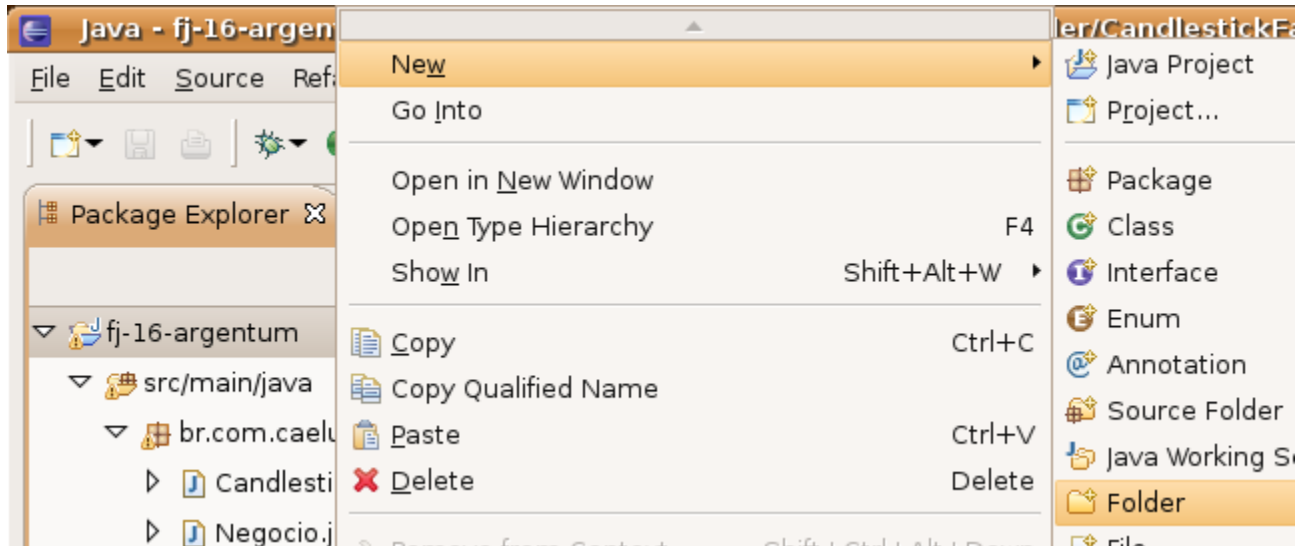
Ou então, se precisarmos processar as anotações de apenas uma única classe, basta indicá-la, como abaixo:

```
stream.processAnnotations(Negocio.class);
```

1.5 - Exercícios: Lendo o XML

1. Para usarmos o XStream, precisamos copiar seus JARs para o nosso projeto e adicioná-los no Build Path. Para facilitar, vamos criar uma pasta **lib** para colocar todos os arquivos JAR que necessitarmos.

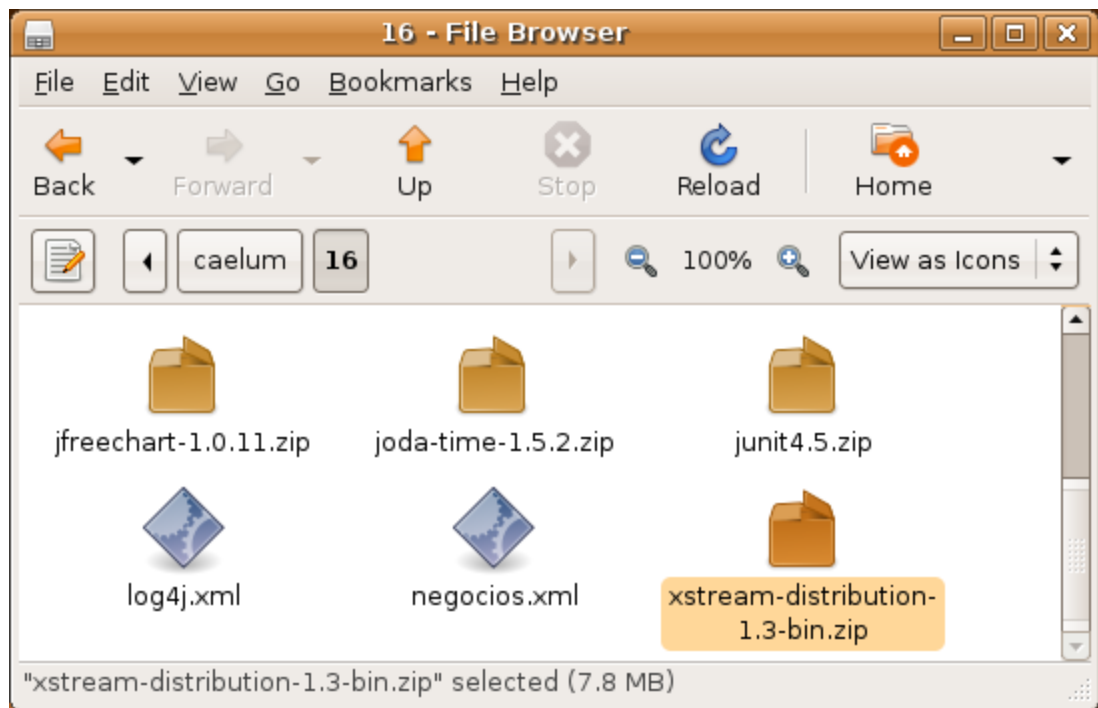
Clique com o botão direito no nome do projeto e vá em **New > Folder**:



Coloque o nome de **lib** e clique OK:

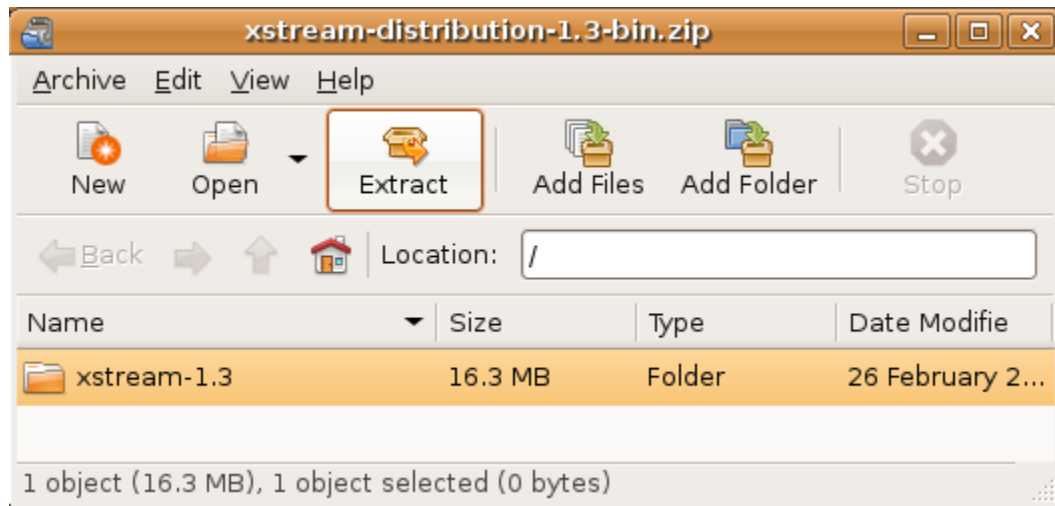


2. Vamos instalar o XStream no nosso projeto. Vá na pasta **Caelum** no seu Desktop e entre em **16**. Localize o arquivo do xstream:

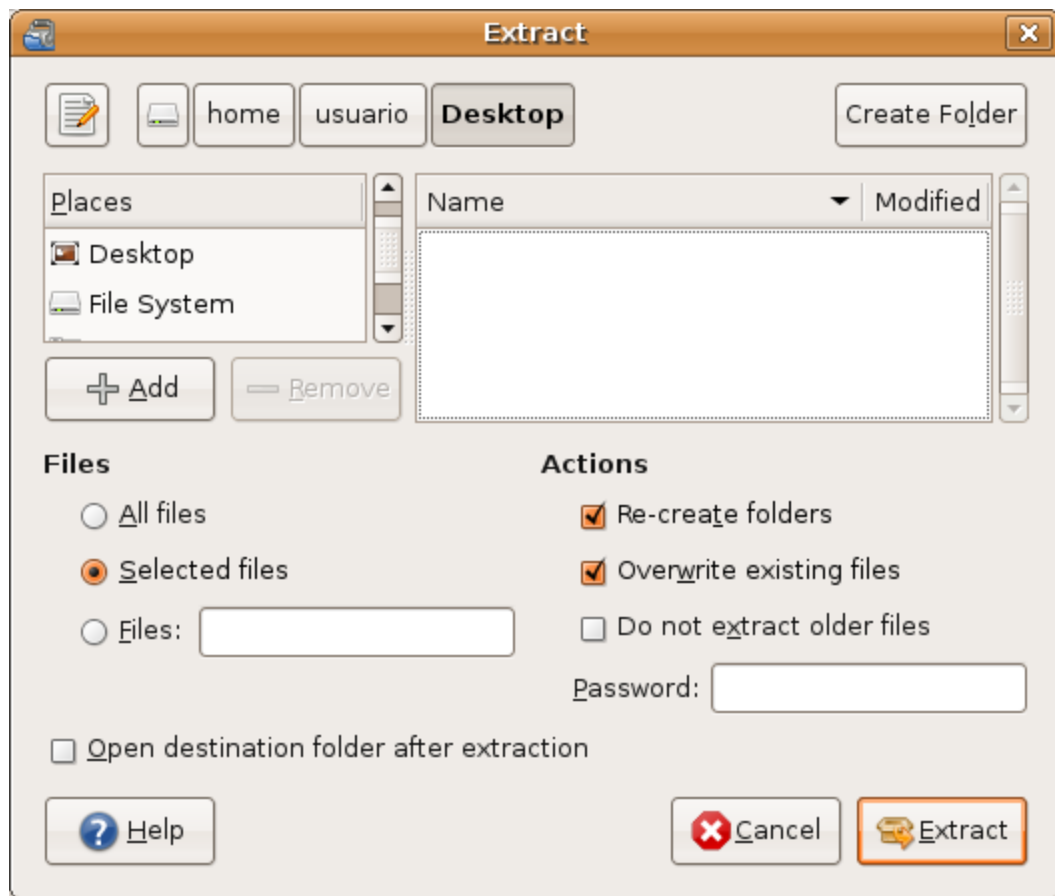


Esse é o mesmo arquivo que é baixado do site do XStream.

Copie esse arquivo para o seu Desktop. Dê dois cliques para abrir o descompactador do Linux. Na próxima tela, clique em Extract:



Com o próprio Desktop selecionado, clique Extract:



3. Entre na pasta **xstream-1.3/lib** no seu Desktop. Há vários JARs do XStream lá, vamos precisar apenas do **xstream-1.3.jar**. O restante são dependências que não precisamos. Copie esse JAR para dentro da sua pasta **lib** do projeto.

Depois, pelo Eclipse, entre na pasta **lib**, de refresh nela, selecione o jar, clique com o botão direito e vá em **Build Path, Add to build path**. A partir de agora o Eclipse considerará as classes do XStream para esse nosso projeto.

4. Vamos agora, finalmente, implementar a leitura do XML, delegando o trabalho para o XStream. Criamos a classe `LeitorXML` dentro do pacote `br.com.caelum.argentum.reader`:

```
package br.com.caelum.argentum.reader;  
  
// imports...
```

```

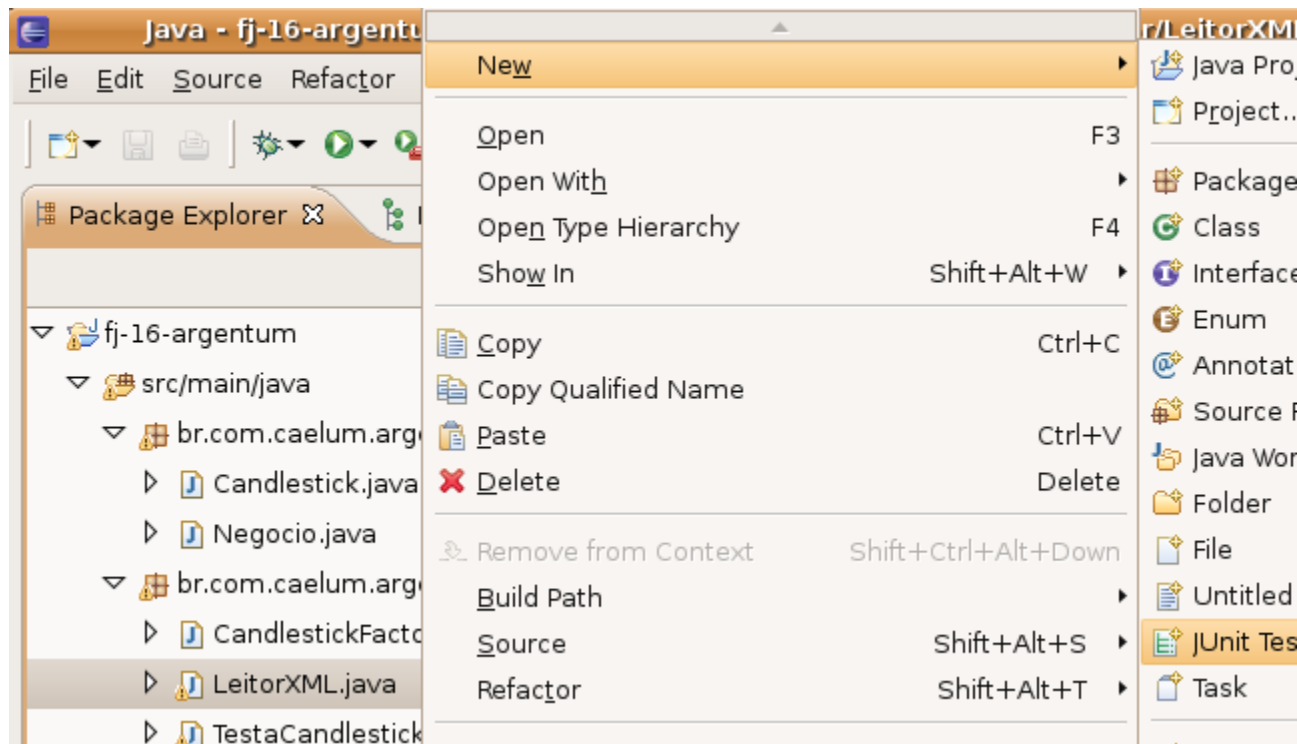
public class LeitorXML {

    public List<Negocio> carrega(Reader fonte) {
        XStream stream = new XStream(new DomDriver());
        stream.alias("negocio", Negocio.class);
        return (List<Negocio>) stream.fromXML(fonte);
    }

}

```

5. Crie um teste unitário `LeitorXMLTest` pelo Eclipse para testarmos a leitura do XML. Basta clicar com botão direito na classe `LeitorXML` e selecionar **New > JUnit Test Case**:



Lembre-se de colocá-lo no diretório **src/test/java**.

Para não ter de criar um arquivo XML no sistema de arquivos, podemos usar um truque interessante: coloque o trecho do XML em uma String Java, e passe um `StringReader`:

```

@Test
public void testLeitorDeXmlCarregaListaDeNegocio() {
    String xmlDeTeste = "..."; // o XML vai aqui!
}

```

```

        LeitorXML leitor = new LeitorXML();
        List<Negocio> negocios = leitor.carrega(new StringR
    }

```

Use o seguinte XML de teste, *substituindo as reticências no código acima pelo texto abaixo*:

```

<list>
  <negocio>
    <preco>43.5</preco>
    <quantidade>1000</quantidade>
    <data>
      <time>555454646</time>
    </data>
  </negocio>
</list>

```

Faça algumas asserções:

- a lista devolvida deve ter tamanho 1
- o negócio deve ter preço 43.5
- a quantidade deve ser 1000

Opcionalmente crie mais de um teste se possível, testando casos excepcionais, como XML inválido, zero negócios, e dados faltando.

6. (importante, conceitual) E o que falta agora? Testar nosso código com um real XML?

É muito comum sentirmos a vontade de fazer um teste "maior": um teste que realmente abre um `FileReader`, passa o XML para o `LeitorXML` e depois chama a `CandlestickFactory` para quebrar os negócios em candles.

Esse teste seria quase um teste de integração, e não de unidade. Se criássemos esse teste, e ele falhasse, ficaria muito difícil de detectar o ponto de falha! Mais ainda: pode parecer difícil de perceber, mas já que testamos todas as peças (units) do nosso motor (aplicação), é muito provável que ele funcione quando rodado! Só falta testar as "ligas", que normalmente são bem leves e poucas (o tal do baixo acoplamento).

Pensar em sempre testar as menores unidades possíveis força que a gente sempre crie programas com baixo acoplamento:

poderíamos ter criado um `LeitorXML` que devolvesse diretamente uma `List`, já chamando a fábrica. Mas isso faria com que o nosso teste do leitor de XML testasse muito mais que apenas a leitura de XML (pois estaria passando pela nossa `CandlestickFactory`).

7. (opcional) Faça mais testes de unidade de outros casos com o `LeitorXML`. Algumas ideias:
 - Teste a leitura de XMLs com quantidades diferentes de negócios (1, 2, nenhum);
 - Teste a passagem de um XML com erros. O que deve acontecer? Uma `ConversionException` do `XStream` ocorre. Pense se seria interessante tratá-la dentro do `Leitor` ou não.
 - Teste a geração do XML (toXML) a partir da lista de negócios lida.