02

Conversores complexos

Esse capítulo da continuidade ao projeto terminado anteriormente. Se você ainda não o possui configurado no Eclipse, baixe e importe (https://s3.amazonaws.com/caelum-online-public/XSTREAM/xstream-fim-do-5.zip) o projeto agora mesmo.

Mas e em casos mais complexos? Vamos criar um conversor completamente distinto para nossa Compra agora. Queremos fazer uma saída:

```
<compra estilo="novo">
  <id>15</id>
  <fornecedor>guilherme.silveira@caelum.com.br</fornecedor>
  <endereco>
    <linha1>Rua Vergueiro 3185</linha1>
    <linha2>8 andar - Sao Paulo - SP</linha2>
  </endereco>
  <produtos>
    <livro codigo="1589">
      <nome>O Pássaro Raro</nome>
      <preco>100.0</preco>
      <descrição>dez histórias sobre a existência</descrição>
    </livro>
    <musica codigo="1590">
      <nome>Meu Passeio</nome>
      <preco>100.0</preco>
      <descrição>música livre</descrição>
    </musica>
  </produtos>
</compra>
```

Para isso adicionaremos um novo teste no nosso CompraTest:

```
@Test
public void deveUsarUmConversorDiferente() {
}
```

Criamos uma compra e pedimos para o xstream registrar um conversor que criaremos:

```
Compra compra = compraComLivroEMusica();

XStream xstream = xstreamParaCompraEProduto();
xstream.registerConverter(new CompraDiferenteConverter());
String xmlGerado = xstream.toXML(compra);

assertEquals(xmlEsperado, xmlGerado);
```

O nosso converter CompraDiferenteConverter sabe lidar com Compra.class, portanto implementamos o método canConvert adequadamente:

```
public class CompraDiferenteConverter implements Converter {

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

No instante de serializar marshal, vamos criar um atributo chamado "estilo", com valor "novo":

```
public void marshal(Object object, HierarchicalStreamWriter writer,
        MarshallingContext context) {
    writer.addAttribute("estilo", "novo");
}
```

Agora precisaremos da compra para poder colocar o id dela. Pegamos o objeto que nos foi passado e fazemos o cast:

```
Compra compra = (Compra) object;
```

Adicionamos uma nova tag:

```
writer.startNode("id");
```

Pedimos para o contexto atual do xstream converter o id da compra com o conversor adequado que ele possui:

```
context.convertAnother(compra.getId());
```

Note que precisamos gerar o getter da Compra! Já aproveitamos e geramos tanto o getter do id quanto dos produtos que usaremos para serializar os produtos com nosso conversor. Terminamos finalizando a tag atual (id):

```
writer.endNode();
```

Ficamos então com o código para a tag "id":

```
writer.startNode("id");
context.convertAnother(compra.getId());
writer.endNode();
```

Desejamos adicionar também a tag "fornecedor", com o valor "guilherme.silveira":

```
writer.startNode("fornecedor");
writer.setValue("guilherme.silveira@caelum.com.br");
writer.endNode();
```

Da mesma maneira que criamos a tag "fornecedor", podemos criar as tags "endereco", "linha1" e "linha2":

```
writer.startNode("endereco");
writer.startNode("linha1");
writer.setValue("Rua Vergueiro 3185");
writer.endNode();
writer.startNode("linha2");
writer.setValue("8 andar - Sao Paulo - SP");
writer.endNode();
writer.endNode();
```

Da mesma maneira que fizemos com o id, serializaremos a lista de produtos com o conversor padrão do Xstream:

```
writer.startNode("produtos");
context.convertAnother(compra.getProdutos());
writer.endNode();
```

Agora o teste passa, uma vez que o xml gerado é exatamente o que esperávamos: usamos o writer para criar um atributo, diversas tags e serializar utilizando os conversores padrão (delegando para o xstream novamente parte da serialização).

No caso da deserialização, adicionamos a validação no mesmo teste:

```
Compra deserializada = (Compra) xstream.fromXML(xmlGerado);
assertEquals(compra, deserializada);
```

E agora precisamos implementar o método unmarshal adequadamente:

```
public Object unmarshal(HierarchicalStreamReader reader,
        UnmarshallingContext context) {
    return null;
}
```

Para ler um atributo bastaria chamar o getAttribute:

```
String estilo = reader.getAttribute("estilo");
```

Entramos na primeira tag de nossa compra, a tag id:

```
reader.moveDown();
```

Agora podemos ler tanto o nome da tag, quanto o valor dela:

```
String nomeId = reader.getNodeName();
String valorId = reader.getValue();
```

Por fim, terminamos de ler a tag id, voltando a tag compra, mas agora já tendo terminado o id:

```
reader.moveUp();
```

Ficamos então por enquanto com a leitura do atributo e do id:

```
String estilo = reader.getAttribute("estilo");
reader.moveDown();
String nomeId = reader.getNodeName();
String valorId = reader.getValue();
reader.moveUp();
```

Da mesma maneira, podemos ler o valor do fornecedor:

```
reader.moveDown();
String fornecedor = reader.getValue();
reader.moveUp();
```

Vamos ler agora a tag endereco (tabs para facilitar a leitura):

```
reader.moveDown();
    String nomeTagEndereco = reader.getNodeName();
    reader.moveDown();
        String linha1 = reader.getValue();
    reader.moveUp();
    reader.moveDown();
        String linha2 = reader.getValue();
    reader.moveUp();
reader.moveUp();
```

Por fim, desejamos converter agora com o convertAnother toda a lista de produtos. Mas note que o convertAnother precisa do objeto atual, para saber qual conversor utilizar. Portanto vamos criar a compra com tudo o que temos até agora:

```
List<Produto> produtos = new ArrayList<>();
int id = Integer.parseInt(valorId);
Compra compra = new Compra(id, produtos);
```

Vamos agora entrar na tag (nó) produtos e convertê-los:

```
reader.moveDown();
List<Produto> produtosConvertidos = (List<Produto>) context.convertAnother(compra, List.class);
```

Por fim, vamos adicionar os produtos convertidos na nossa lista de produtos que passamos no construtor da compra, e fechar o nó:

```
produtos.addAll(produtosConvertidos);
reader.moveUp();
```

Pronto. O que fizemos? Um conversor que ao deserializar, lê um atributo, visita as tags (os nós) usando moveDown e volta para o nó principal com o moveUp, instancia uma Compra e usa o contexto para deserializar os produtos:

```
String estilo = reader.getAttribute("estilo");

reader.moveDown();
String nomeId = reader.getNodeName();
String valorId = reader.getValue();
reader.moveUp();

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

reader.moveDown();
String nomeTagEndereco = reader.getNodeName();
reader.moveDown();
String linha1 = reader.getValue();
reader.moveUp();
reader.moveDown();
String linha2 = reader.getValue();
reader.moveUp();
reader.moveUp();

List<Produto> produtos = new ArrayList<>();
int id = Integer.parseInt(valorId);
Compra compra = new Compra(id, produtos);

reader.moveDown();
List<Produto> produtosConvertidos = (List<Produto>) context.convertAnother(compra, List.class);
produtos.addAll(produtosConvertidos);
reader.moveUp();

return compra;
```

Note que usamos a conversão para int nossa: o convertAnother precisava da compra. Por padrão o XStream, assim como diversas outras bibliotecas, tem o uso facilitado se existe um construtor sem argumentos e os setters - o que não é obrigatório, mas facilita a criação dos conversores.