01

Mapeando XML para um objeto

Transcrição

Já aprendemos três formas diferentes de ler arquivos XML, como realizar pesquisas nesses arquivos e como convertê-los para outros formatos. Para todos esses procedimentos, precisamos escrever códigos que executassem aquilo que desejávamos. Mas será que não existe uma forma ainda mais simples de conseguir uma cópia fiel de um XML na memória, fazendo apenas um mapeamento, sem criar uma classe, percorrer uma lista ou algo do gênero?

Pensando nisso, surgiu o JAXB, ou "Java API for XML Binding", uma especificação do Java que consegue associar um XML diretamente a uma classe sem precisar escrever código que faça tal mapeamento. Para começarmos a trabalhar com essa especificação, criaremos uma classe `Venda.java` no nosso sistema, já que esta é a principal tag do nosso arquivo XML. Assim como `Produto`, essa classe será armazenada no pacote `Model`.

Uma venda possui uma `formaDePagamento`, que é um texto, e diversos produtos. Sendo assim, criaremos um string `formaDePagamento` e uma lista de `Produto` chamada `produtos`, além dos seus getters e setters.

```
public class Venda {

    private String formaDePagamento;
    private List<Produto> produtos;

    public String getFormaDePagamento() {
        return formaDePagamento;
    }
    public void setFormaDePagamento(String formaDePagamento) {
        this.formaDePagamento = formaDePagamento;
    }
    public List<Produto> getProdutos() {
        return produtos;
    }
    public void setProdutos(List<Produto> produtos) {
        this.produtos = produtos;
    }

}
```

Agora precisamos de um código que mapeie o XML com a nossa classe `Venda`. Para isso, no pacote `Teste`, criaremos uma nova classe `MapeiaXMLDireto` que possui um método `main()`. Para o mapeamento, usaremos a classe `JAXBContext`, cuja instanciação recebe a classe que será mapeada - no caso, `Venda.class`. Armazenaremos essa instância em uma variável local `jaxbContext`, faremos as importações necessárias e adicionaremos uma `Exception` ao método.

```
public class MapeiaXMLDireto {
    public static void main(String[] args) throws Exception {
        JAXBContext jaxbContext = JAXBContext.newInstance(Venda.class);
    }

}
```

A classe responsável por fazer o caminho de XML para um objeto é chamada de `Unmarshaller`. Por meio da nossa variável `jaxbContext`, podemos acessar o método `createUnmarshaller()`, criando um objeto desse tipo que associaremos a uma nova variáverl `unmarshaller`. A partir desta variável, chamaremos o método `unmarshall()` passando como argumento o nosso arquivo XML. Associaremos o retorno desse método a uma variável local `venda` do tipo `Venda`. Como o método `unmarshall()` retorna um `Object`, faremos um casting. Por fim, faremos um `System.out.print()` da nossa `venda`.

```
public class MapeiaXMLDireto {
    public static void main(String[] args) throws Exception {
        JAXBContext jaxbContext = JAXBContext.newInstance(Venda.class);
        Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();

        Venda venda = (Venda) unmarshaller.unmarshal(new File("src/vendas.xml"));
        System.out.println(venda);
    }

}
```

Executando o código dessa forma, teremos como retorno um erro, pois o `Unmarshaller` não estava esperando um elemento `venda`. Para corrigirmos esse erro, precisaremos informar que a classe `Venda` representa um XML com a instrução `@XmlRootElement`.

```
@XmlRootElement
public class Venda {

    private String formaDePagamento;
    private List<Produto> produtos;

    public String getFormaDePagamento() {
        return formaDePagamento;
    }
    public void setFormaDePagamento(String formaDePagamento) {
        this.formaDePagamento = formaDePagamento;
    }
    public List<Produto> getProdutos() {
        return produtos;
    }
    public void setProdutos(List<Produto> produtos) {
        this.produtos = produtos;
    }

}
```

Se executarmos novamente nosso código, teremos como retorno:

```
br.com.alura.Model.Venda@119d7047
```

Arrumaremos essse retorno adicionando um método sobrescrito `toString()` à nossa `Venda`. Nele, retornaremos o texto `"Forma de pagamento: "` associado ao atributo `formaDePagamento`, além de `"\n produtos: "` (com `\n` para pularmos uma linha) concatenado com os nossos `produtos`.

```
@Override
public String toString() {
    return "Forma de pagamento: " + formaDePagamento + "\n produtos: " + produtos;
}
```

Executando novamente o código, teremos:

```
Forma de pagamento: Débito
 produtos: [ Nome:null
 Preço:0.0
]
```

Conseguimos associar o atributo `formaDePagamento`, mas os nossos produtos vieram com o nome nulo (`null`) e o preço `0.0`, o valor padrão de um double. Isso aconteceu porque a `List<Produto> produtos` tem uma tag que a representa. Sendo assim, adicionaremos uma anotação `@XmlElementWrapper()` passando a instrução `name="produtos"`.

```
@XmlElementWrapper(name="produtos")
private List<Produto> produtos;
```

Executando novamente, teremos uma `Exception` lançada, pois o Java se confunde ao encontrar a propriedade `produtos` e um getter/setter com o mesmo nome. Para contornarmos tal problema, adicionaremos a anotação `@XmlAccessorType(XmlAccesType.FIELD)`, definindo que a forma de acessar os atributos é pelo campo.

```
@XmlRootElement
@XmlAccessorType(XmlAccesType.FIELD)
public class Venda {
//...
}
```

Dessa vez a execução do código trará uma lista vazia:

```
Forma de pagamento: Débito
 produtos:[]
```

Novamente corrigiremos o problema com uma anotação - no caso, `@XmlElement(name="produto")`, indicando que a tag `produto` é o elemento que engloba cada item.

```
@XmlElementWrapper(name="produtos")
@XmlElement(name="produto")
private List<Produto> produtos;
```

Finalmente, a execução do código nos trará a lista de produtos preenchida:

```
Forma de pagamento: Débito
 produtos:[ Nome:Livro de xml
 Preço:29.9
,  Nome:Livro de O.O. java
```

```
 Preço:29.9
]
```

Ou seja, conseguimos mapear o nosso XML diretamente para uma classe venda. No próximo vídeo aprenderemos a fazer o processo contrário, transformando um objeto em um XML.