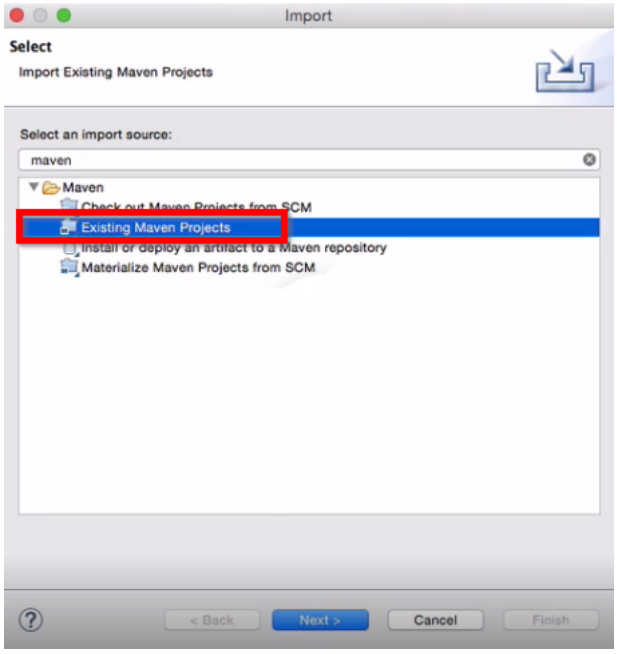
01

Integrando Maven e Eclipse

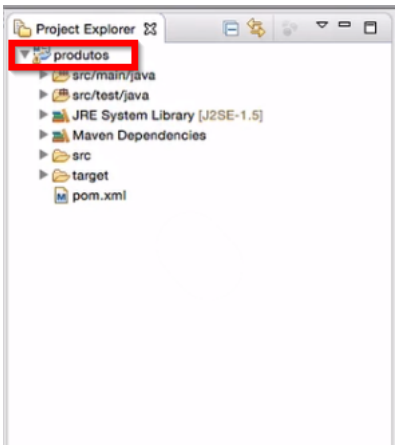
Transcrição

Nesta aula traremos o projeto para o Eclipse, utilizando a versão *Eclipse Mars*. Com o programa aberto, selecionaremos o diretório de *workspace* e importaremos um projeto do Maven. No cabeçalho, selecionaremos as opções "File > Import". Surgirá uma caixa de diálogo, e na opção "Select an import source" buscaremos a opção "Maven", depois selecionaremos o item `Existing Maven Projects` na pasta `Maven`.



Feito isso, escolheremos o diretório que abriga nosso projeto, no caso, `Produtos`. O Eclipse irá checar se o arquivo que queremos importar é o `pom.xml`, o que é correto, portanto pressionaremos o botão "Finish".

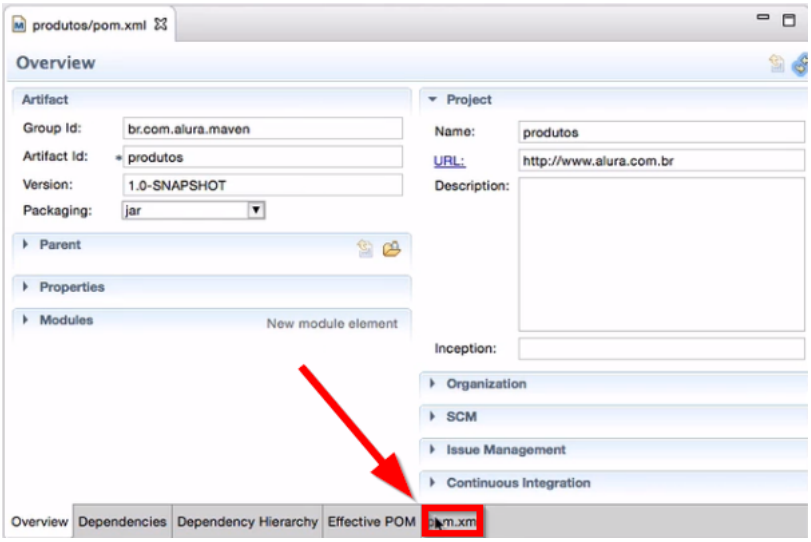
Quando importamos um projeto do Maven para o Eclipse pela primeira vez, alguns plugins são baixados automaticamente, por isso o processo pode ser um pouco mais lento. No momento em que a importação é finalizada, clicaremos sobre a opção "Workbench" na parte superior direita da tela, e seremos direcionados para a área de trabalho do Eclipse, com o projeto pronto para ser trabalhado.



Na área "Project Explorer" teremos o projeto `produtos`, que contém o diretório `src/main/java`, que por sua vez armazena os arquivos `App.java` e `Produto.java`, isto é, as duas classes. Há também os arquivos de teste, com um

pacote `br.com.alura.maven` e a classe `AppTest.java`, e as dependências do Maven em `Maven Dependencies` como o JUnit.

Ao final teremos o arquivo `pom.xml` que, ao ser selecionado, nos dá acesso às opções que permitem que configuremos o arquivo visualmente. Clicaremos sobre "pom.xml" para trabalharmos nas linhas de código.



Temos o projeto inteiro no Eclipse, configurado e executado de forma correta. Vamos começar a explorar os arquivos. Clicaremos sobre a classe `Produto.java`, armazenada no diretório `src/main/java`:

```
package br.com.alura.maven;

public class Produto {

}
```

Faremos algumas alterações nesta classe — incluiremos os campos `nome`, que é `private String`, o campo `preco`, e `private double`.

```
package br.com.alura.maven;

public class Produto {

    private String nome;
    private double preco;

}
```

Os dois campos serão recebidos no construtor, portanto vamos adicioná-lo. Utilizaremos o atalho "Ctrl + 3" para acessar o "Quick Access", e com "generate" buscamos a opção "Generate Constructor using Fields", isto é, "Gerar construtor usando campos", que no caso serão `nome` e `preco`. Uma nova caixa de diálogo surgirá, e nós devemos justamente marcar as opções de campos "nome" e "preco", pressionar o botão "OK", com que o construtor será gerado.

```
package br.com.alura.maven;

public class Produto {
```

```
    private String nome;
    private double preco;

    public Produto(String nome, double preco) {
        super();
        this.nome = nome;
        this.preco = preco;
    }

}
```

A classe `Produto` não permite alteração, ou seja, não podemos modificar o `nome` de um produto, nem seu `preco`. De acordo com as boas práticas, se temos um campo que não é alterável incluímos a variável `final`, pois desse modo garantimos que os valores sejam exatamente os mesmos passados na construção.

```
package br.com.alura.maven;

public class Produto {

    private final String nome;
    private final double preco;

    public Produto(String nome, double preco) {
        super();
        this.nome = nome;
        this.preco = preco;
    }

}
```

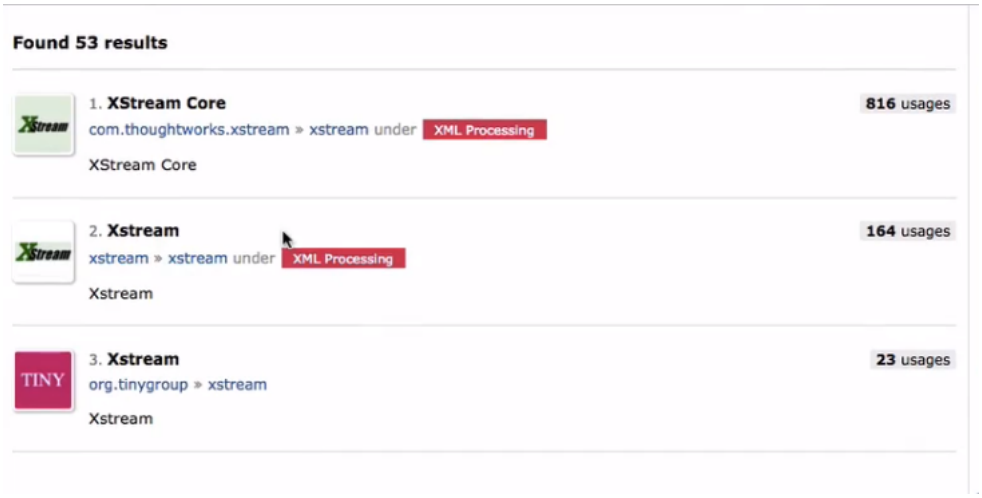
Geramos a classe `Produto`, salva automaticamente. Caso acessemos nosso diretório pelo terminal e solicitemos a compilação do Maven, ela será executada normalmente fora do Eclipse, mas podemos realizar a compilação no editor. Veremos essa interação entre Maven e Eclipse adiante.

O Eclipse é capaz de importar um projeto Maven, mas e se quisermos fazer alterações nesse projeto? No momento estamos utilizando apenas o JUnit como dependência, e incluiremos outra, a XStream.

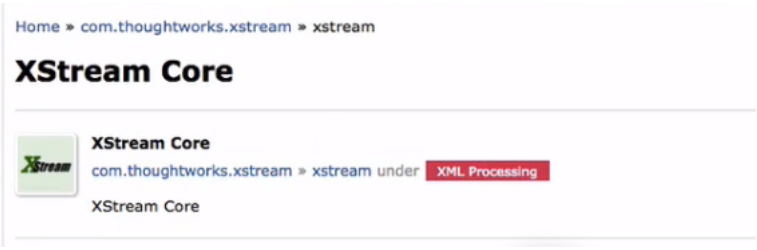
No arquivo `pom.xml`, mais precisamente entre as tags de `<dependency>`, adicionaremos uma nova dependência. Precisamos incluir o `<groupId>`, e caso não tenhamos essa informação basta procurá-la no Google. Vamos procurar por "XStream Maven".

Muitas vezes, ao buscarmos Maven associado a alguma biblioteca, encontraremos o site Maven Repository (https://mvnrepository.com/), que usaremos em nosso dia a dia para procurar bibliotecas. Existe um mecanismo de busca interna, e nós procuraremos por "XStream".

Existem várias opções disponíveis, alguns projetos que fazem uso dessa biblioteca, e que podem ser baixados diretamente no site. Contudo, estamos buscando pela versão original do XStream.



Clicaremos sobre a primeira opção, "XStream Core". Verificaremos que a última versão disponível dessa biblioteca é a "1.4.8". Para "XStream" a última versão é "1.2.2". Até a versão `1.2.2` , a biblioteca era lançada com o nome "XStream Xstream". A partir daí, a biblioteca foi lançada como "com.thoughtworks.xstream", e este é nosso *group Id*.



Quando procuramos uma biblioteca a ser adicionada como dependência para o Maven, é recomendado o *Maven Repository*, pois conseguimos conhecer as suas variações disponíveis. É extremamente comum haver alterações no padrão das bibliotecas quando elas começam a se desenvolver mais.

Seguindo, escreveremos as informações necessárias no arquivo `pom.xml` :

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.com.alura.maven</groupId>
    <artifactId>produtos</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>produtos</name>
    <url>http://www.alura.com.br</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>jnuit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>

    <dependency>
        <groupId>com.thoughtworks.xstream</groupId>
    </dependency>
```

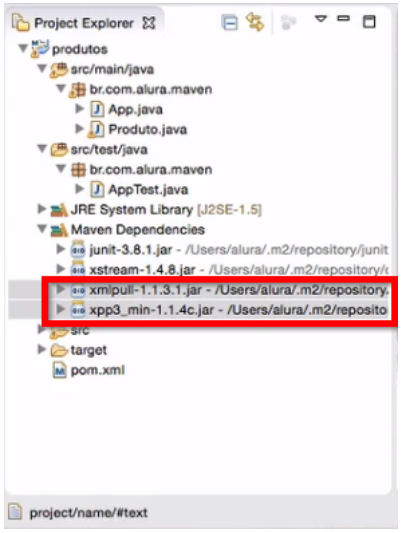
```
    </dependencies>
</project>
```

Há uma questão que pode atrapalhar a implementação da dependência — digitamos o `<groupId>` diretamente no código, e isso pode gerar erros de digitação, afinal são muitas palavras estrangeiras, e não exatamente simples. Para evitar problemas dessa natureza, acessaremos o *Maven Repository*, clicamos na versão da biblioteca que utilizaremos — no caso, `1.4.8` —, copiaremos o código correspondente à dependência:

```
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.4.8</version>
</dependency>
```

E colaremos o código em `pom.xml`. Como não mencionamos o escopo, estamos aplicando a biblioteca para a aplicação inteira. Aprenderemos mais sobre essa questão mais adiante. Salvaremos as modificações, e o Eclipse exibirá a informação "build workspace" na parte inferior direita da tela. Isso ocorre porque o XStream foi baixando no diretório `Maven Dependencies`, e já está incluso no *classpath*.

Nota-se que foram disponibilizados dois arquivos diferentes no `Maven Dependencies`, são eles `xmipull-1.1.3.1.jar` e `xpp3_min-1.1.4c.jar`.



Os dois arquivos são opcionais para o XStream, ainda que muito utilizados. No momento em que realizamos o download do XStream, havia a opção em `.zip`, com mais arquivos, ou a versão em `.jar`, e optamos pela segunda alternativa. Caso tentássemos utilizar algumas features do XStream, teríamos alguns problemas em tempo de execução.

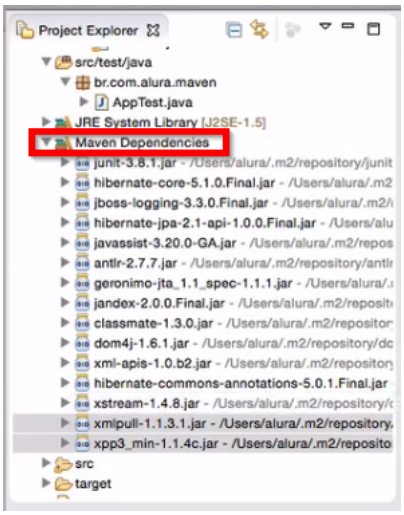
O Maven é extremamente inteligente nesse sentido, e no momento em que inserimos a nova dependência e salvamos o arquivo, há um diálogo entre o editor e a ferramenta, de forma que novos conteúdos foram baixados automaticamente. Seria o equivalente a acionarmos `mvn compile` no terminal.

Incluímos apenas a dependência do XStream em nosso código do `pom.xml`, porém, para o XStream ser compilado, ele precisa de muitos recursos, os quais podemos verificar no *Maven Repository*. Existe uma série de pacotes no `.jar` do XStream, sendo que alguns de compilação estão configurados como necessários.

O mesmo procedimento se dará caso baixemos outras dependências, como o *Hibernate*. No *Maven Repository* buscaremos por "Hibernate" e implementaremos a versão `5.1.0 Final` padrão, copiando o seguinte código e colando-o no arquivo `pom.xml` :

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.1.0.Final</version>
</dependency>
```

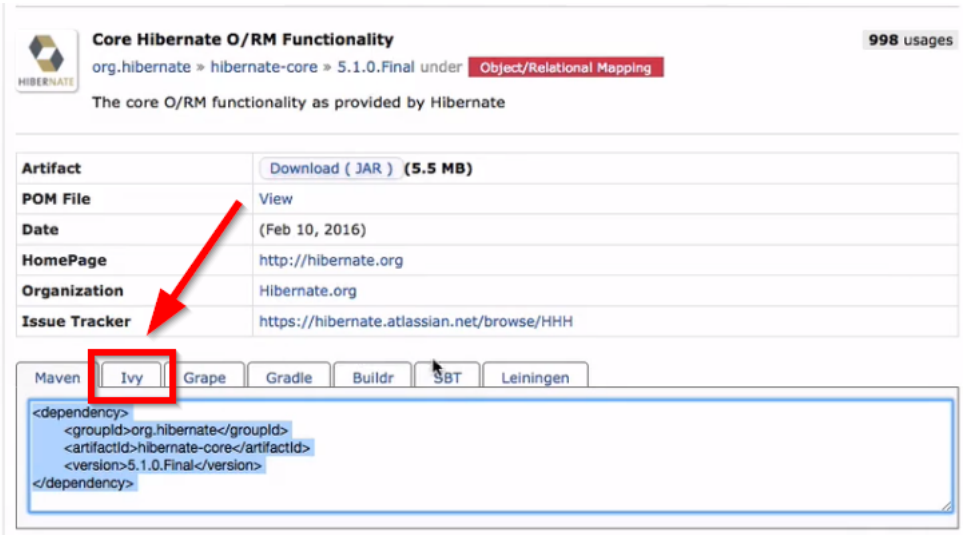
Ao salvarmos o arquivo no Eclipse, várias dependências obrigatórias serão baixadas, e todas elas estarão disponíveis no *classpath*. Isso porque o Maven faz uma varredura na árvore de dependências, realizando os downloads necessários.



Além do *build*, o Maven gerencia as dependências. Podemos utilizar essa ferramenta apenas para o gerenciamento de dependências ou para o processo de *build*, e ainda existe a possibilidade de usarmos apenas a parte de dependências sem o Maven em si.

Quando vamos baixar uma biblioteca no *Maven Repository*, existe a opção "Ivy" na caixa de texto em que é disponibilizado o código, responsável pela a parte de dependências.

> Trata-se de outra ferramenta que gerencia apenas dependências. Na Alura temos um [curso exclusivo sobre Ivy. (https://cursos.alura.com.br/course/ivy)](https://cursos.alura.com.br/course/ivy)



Há outras opções de ferramentas, como Grape, Gradle, Apache Buildr, SBT e Leiningen, que servem para diversas linguagens, mas todas essas fazem seu *build* em cima da estrutura criada pelo Maven.

Até este ponto entendemos como o Eclipse interage com o Maven todas as vezes em que adicionamos ou eliminamos uma dependência. No caso de exclusão, o projeto é atualizado sem deixar resíduos da biblioteca. Uma vez que importamos um projeto do Maven no Eclipse as configurações são realizadas, facilitando muito o nosso trabalho.