Introduction to Apache Maven 2

Sing Li

December 19, 2006

Modern software projects are no longer solely monolithic creations of single local project teams. With the increased availability of robust, enterprise-grade open source components, today's software projects require dynamic collaboration among project teams and often depend on a mix of globally created and maintained components. Now in its second generation, the Apache Maven build system -- unlike legacy build tools created before the Internet-enabled era of global software development -- was designed from the ground up to take on these modern challenges. This tutorial gets you started with Maven 2.

Before you start

Modern software development based on robust, enterprise-grade open source technologies requires a new breed of build and project collaboration tool. The engine at the core of Apache Maven 2 works to simplify building and managing large and often complex collaborative software projects. Yet Maven 2's design aims to be friendly even to developers unfamiliar with the challenges of working in large project team environments. Focusing initially on the beginner single developer, this tutorial gradually introduces some of the collaborative concepts and features that are available with Maven 2. You are encouraged to build on the introduction this tutorial provides by exploring the advanced features of Maven 2 that are beyond its scope.

About this tutorial

This tutorial guides you step-by-step through the fundamental concepts and hands-on exercises with Maven 2:

- Overview of Maven 2
- Understanding the Maven 2 dependency management model
- Maven 2 repository and Maven 2 coordinates
- Maven 2 life cycles, phases, plug-ins, and mojos
- Downloading and installing Maven 2
- Hands-on Maven 2 -- your first Maven 2 project
- Customizing the project object model (POM)
- Working with multiple projects
- Hands-on Maven 2 -- working with multiple project builds
- Installing the Maven 2.x Plug-in for Eclipse 3.2

- Working with the Maven 2.x Plug-in for Eclipse 3.2

As you complete this tutorial, you will gain an appreciation and understanding of the philosophy behind the design of Maven 2. Furthermore, you will be familiar with the fundamental skills required to work on projects built using Maven 2. This is a passport to most of the large projects in the Apache and Codehaus communities. Most important, you'll be ready to apply Maven 2 to your daily project build and management activities.

Prerequisites

You should be familiar with Java™ development in general. This tutorial assumes that you understand the value and basic operations of a build tool, including dependency management and output packaging. You need to be able to work with Eclipse 3.2 as an IDE to work through the Maven 2.x Plug-in for Eclipse section. An exposure to large open source projects, such as those under the Apache Software Foundation's management, is highly valuable. An understanding of Java 5 coding, including generics, is helpful. Experience working with various project building technologies such as Ant, `autoconf`, `make`, and `nmake` is beneficial but not mandatory.

System requirements

To follow along and try out the code for this tutorial, you need a working installation of Sun's JDK 1.5.0_09 (or later) or the IBM JDK 1.5.0 SR3.

For the sections on the Maven 2.x Plug-in for Eclipse, you need a working installation of Eclipse 3.2.1 or later.

The recommended system configuration for the tutorial is:

- A system supporting the JDK/JRE mentioned above with at least 1GB of main memory
- At least 20MB of disk space to install the software components and examples

The instructions in the tutorial are based on a Microsoft Windows operating system. All of the tools covered in the tutorial also work on Linux® and UNIX® systems.

Overview of Maven 2

Maven is a top-level open source Apache Software Foundation project, created originally to manage the complex build process of the Jakarta Turbine project. Since this humble beginning, development projects in both the open source and the private realm have embraced Maven as the project build system of choice. Rapidly evolving, and now in version 2, Maven has grown from a customized build tool for a single complex project to a generalized build management system with a cornucopia of features applicable to most software development scenarios.

In a nutshell, Maven 2:

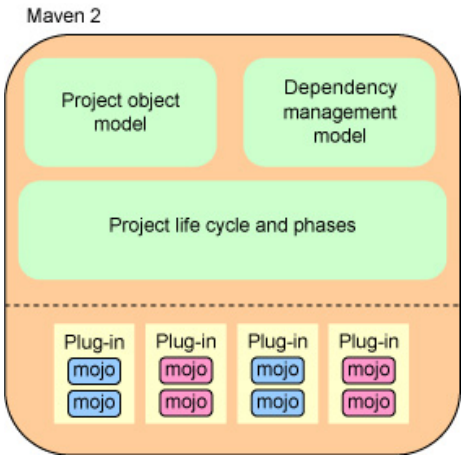
- Understands how a project is typically built.
- Makes use of its built-in project knowledge to simplify and facilitate project builds.

- Leverages its built-in project knowledge to help users understand a complex project's structure and potential variations in the build process.
- Prescribes and enforces a proven dependency management system that is in tune with today's globalized and connected project teams.
- Provides a simple and unintrusive user experience for simple projects, leveraging its internal knowledge.
- Is completely flexible for power users; the built-in models can be overridden and adapted declaratively (via configuration, modification of metadata, or creation of custom plug-ins) for specific application scenarios.
- Is fully extensible for scenario details not yet covered by existing behaviors.
- Is continuously improved by capturing any newfound best practices and identified commonality between user communities and making them a part of Maven's built-in project knowledge.

Maven 2 -- A conceptual overview

To capture project-building knowledge, Maven 2 relies on an evolving set of conceptual models of how things should work. These models, partially hardcoded as part of the Maven 2 code base, are constantly refined through new Maven releases. Figure 1 illustrates the key Maven 2 models:

Figure 1. Maven 2 object and operation models



The key components in Figure 1 are:

- **Project object model (POM):** The POM is a cornerstone model for Maven 2. Part of this model is already built into the Maven engine (fondly called the *reactor*), and you provide other parts declaratively through an XML-based metadata file named pom.xml.
- **Dependency management model:** Maven is particular about how project dependencies are managed. Dependency management is a gray area that typical build-management tools and systems choose not to be specific about. The Maven dependency management model is built into Maven 2 and can be adapted to most requirements. This model is a proven workable and productive model currently deployed by major open source projects.
- **Build life cycle and phases**: Coupled to the POM are the notions of build *life cycle* and *phases*. This is Maven 2's interface between its built-in conceptual models and the real

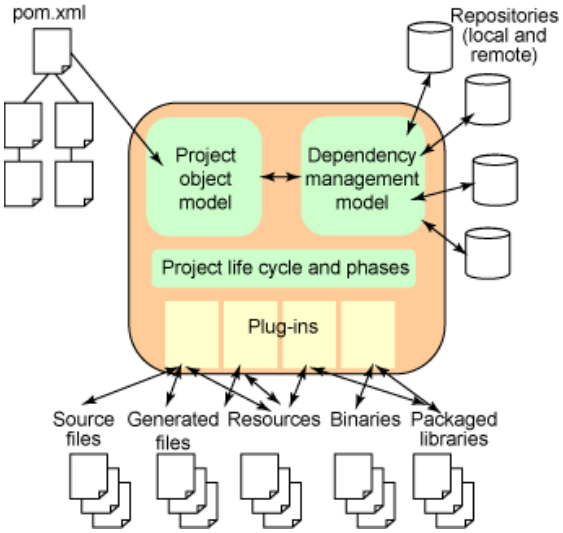
physical world. When you use Maven, work is performed exclusively via plug-ins. Maven 2 orchestrates these plug-ins, following a series of well-defined phases in a build cycle.

Don't worry if some of these concepts still seem a little fuzzy. The following sections provide concrete examples solidifying the concepts behind these models.

Maven 2 -- A physical overview

Figure 2 reveals the operation of and interactions with Maven 2, exposing its physical composition. Figure 2 provides you with a view of how you might interact with Maven 2:

Figure 2. Maven 2 operation and interaction model



In Figure 2, the POM is Maven's understanding of your particular project. This model is formed by declarative descriptions contained in a series of pom.xml files. The pom.xml files form a tree, and each can inherit attributes from its parent. Maven 2 provides a Super POM. The Super POM sits at the top the hierarchy tree and contains default common attributes for all projects; every project POM inherits from it.

Dependencies are specified as part of the pom.xml file. Maven resolves project dependencies according to its dependency management model. Maven 2 looks for dependent components (called *artifacts* in Maven terminology) in local and global repositories. Artifacts resolved in remote repositories are downloaded to the local repository for efficiency of subsequent access. The dependency resolver in Maven 2 can deal with *transitive* dependencies. That is, it works properly when resolving dependencies that your dependencies depend on.

The Maven engine itself performs almost all its file-handling tasks through *plug-ins*. Plug-ins are configured and described in the pom.xml file. The plug-ins themselves are handled as artifacts by the dependency management system and are downloaded on demand as they are needed for a build task. Each plug-in can be associated with the various phases of a life cycle. The Maven engine has a state machine that marches through the life-cycle phases and invokes plug-ins as necessary.

Understanding the Maven 2 dependency management model

You need to understand how the Maven 2 dependency management model works before you can make use of Maven 2 effectively.

The dependency management model is adapted for projects whose software components (called *modules*) might be developed by different project teams. It supports continuous independent development and refinement of all dependent modules.

This team collaboration scenario is the norm with open source projects founded and maintained over the Internet and is becoming more prevalent in corporate circles where in-house development meets the open source or the outsourced world.

Resolving project dependencies

The Maven 2 dependency management engine helps resolve project dependencies during the build process.

Maven local and remote repositories

Your Maven 2 local repository is a directory on your disk, typically located at *HomeDirectory*/.m2/repository. This repository acts as a high-performance local cache, storing any artifacts downloaded as a result of dependency resolution. Remote repositories are accessed over the network. You can maintain a list of remote repositories to use in your settings.xml configuration file.

In practice, dependencies are specified in `<dependencies>` elements within a pom.xml file and are fed into Maven as part of the POM.

Project dependencies are stored on *repository servers* (simply called *repositories* in Maven terminology). Successful dependency resolution depends on finding the required dependent artifact from a repository that contains the artifact.

Maven configuration through settings.xml

You can specify configuration properties that affect Maven operation in a settings.xml file. The default settings file is *MavenInstallationDirectory*/conf/settings.xml. Maven 2 users can maintain *UserHomeDirectory*/.m2/settings.xml to override some configuration properties. See the Maven settings reference for more information on the configurable settings.

Based on the project dependency information in the POM, the dependencies resolver attempts to resolve the dependencies in the following order:

1. Your local repository is checked for the dependency.
2. A list of remote repositories is checked for the dependency.
3. Failing 1 and 2, an error is reported.

By default, the first remote repository contacted in step 2 is a worldwide-accessible centralized Maven 2 repository containing artifacts for most popular open source projects. In the case of in-house development, you can set up additional remote repositories containing release artifacts

from in-house developed modules. The `<repositories>` element in settings.xml can be used to configure these additional remote repositories.

Single copy of artifact enforced

When you use Maven 2 for your project builds, the dependency resolution via a centralized repository ensures that only a single copy of a dependent artifact exists, regardless of how many projects or subprojects reference it. This is a vital property for multimodule project builds because inclusion of multiple copies of artifacts can lead to project consistency and integrity problems.

Repositories and coordinates

Maven 2 repositories store a collection of artifacts used by Maven during dependency resolution for a project. Local repositories are accessed on the local disk, and remote repositories are accessed through the network.

An artifact is usually bundled as a JAR file containing the binary library or executable. This is known as an artifact's *type*. In practice, however, an artifact can also be a WAR, EAR, or other code-bundling type.

Maven 2 takes advantage of an operating system's directory structure for quick indexing of the collection of artifacts stored within a repository. This repository index system relies on the ability to identify any artifact uniquely via its *coordinate*.

Maven coordinates

A Maven coordinate is a tuple of values that uniquely identifies any artifact. A coordinate comprises three pieces of information:

- The **group ID**: The entity or organization responsible for producing the artifact. For example, `com.ibm.devworks` can be a group ID.
- The **artifact ID**: The name of the actual artifact. For example, a project with a main class called `OpsImp` may use `OpsImp` as its artifact ID.
- The **version**: A version number of the artifact. The supported format is in the form of *`mmm.nnn.bbb-qqqqqqq-dd`*, where *`mmm`* is the major version number, *`nnn`* is the minor version number, and *`bbb`* is the bugfix level. Optionally, either *`qqqqq`* (qualifier) or *`dd`* (build number) can also be added to the version number.

Maven coordinates are used throughout Maven configuration and POM files. For example, to specify a project dependency on a module entitled `OpsImp` at the 1.0-SNAPSHOT level, a pom.xml file includes the segment shown in Listing 1:

Listing 1. Maven coordinate for an example OpsImp module

```
<dependencies>
   <dependency>
      <groupId>com.ibm.devworks</groupId>
      <artifactId>OpsImp</artifactId>
      <version>1.0-SNAPSHOT</version>
   </dependency>
</dependencies>
```

The special `SNAPSHOT` qualifier tells Maven 2 that the project or module is under development and that it should fetch the latest copy of the artifact available.

To specify that the project depends on JUnit for unit testing, JUnit 3.8.1's coordinates can be added as a dependency in the project's pom.xml, as shown in Listing 2:

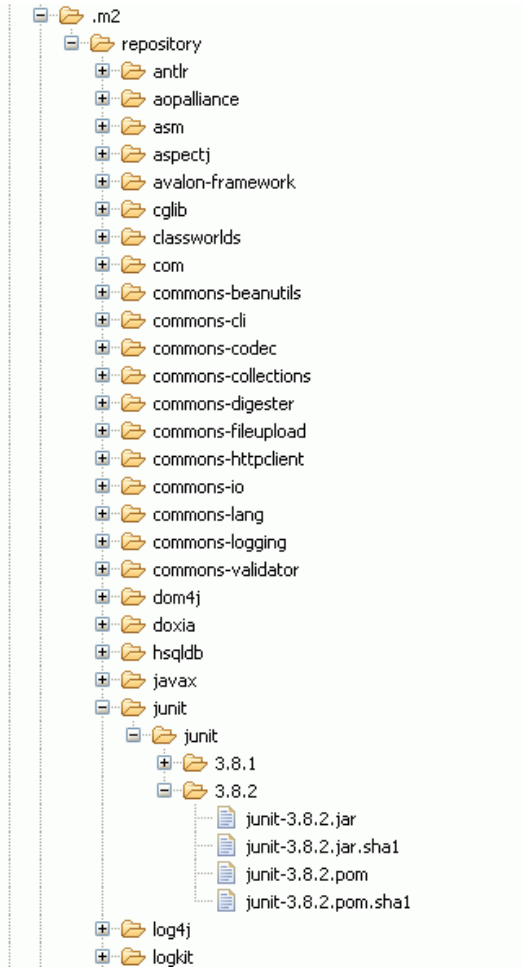
Listing 2. Maven coordinate for a JUnit dependency

```
<dependencies>
   <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
   </dependency>
</dependencies>
```

Looking into a Maven repository

Because Maven repositories are ordinary directory trees, you can readily take a look at how artifacts are stored on disk. Figure 3 is a portion of the local repository, showing the location of the JUnit 3.8.1 artifact:

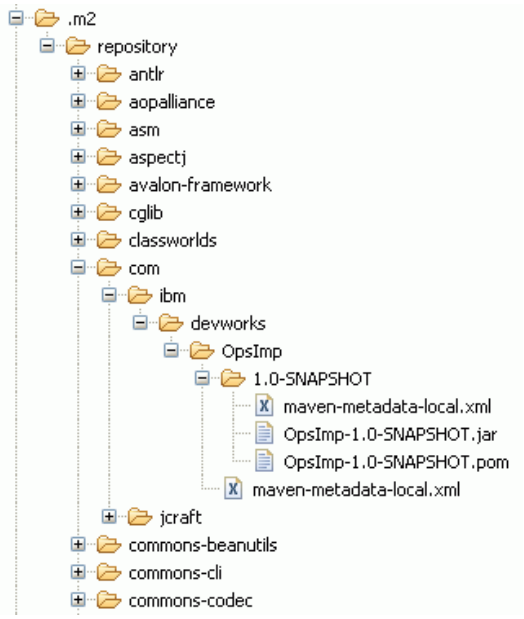
Figure 3. Inside a Maven 2 repository



In Figure 3, you can see that Maven maintains an artifact's POM file, together with checksum hashes for both the artifact and its POM in the repository. These files help ensure artifact integrity when artifacts are transferred between repositories. This artifact has been downloaded from the central repository and placed into the local repository by Maven's dependency management engine.

In Figure 4, the artifact with coordinates `com.ibm.devworks/OpsImp/1.0-SNAPSHOT` is shown in the local repository. The artifact is in the directory together with the POM file. In this case, the artifact is installed locally.

Figure 4. OpsImp artifact in a local repository



Maven 2 life cycles, phases, plug-ins, and mojos

Maven accomplishes most of its build tasks through the action of plug-ins. You can think of the Maven engine as an orchestrator of plug-in actions.

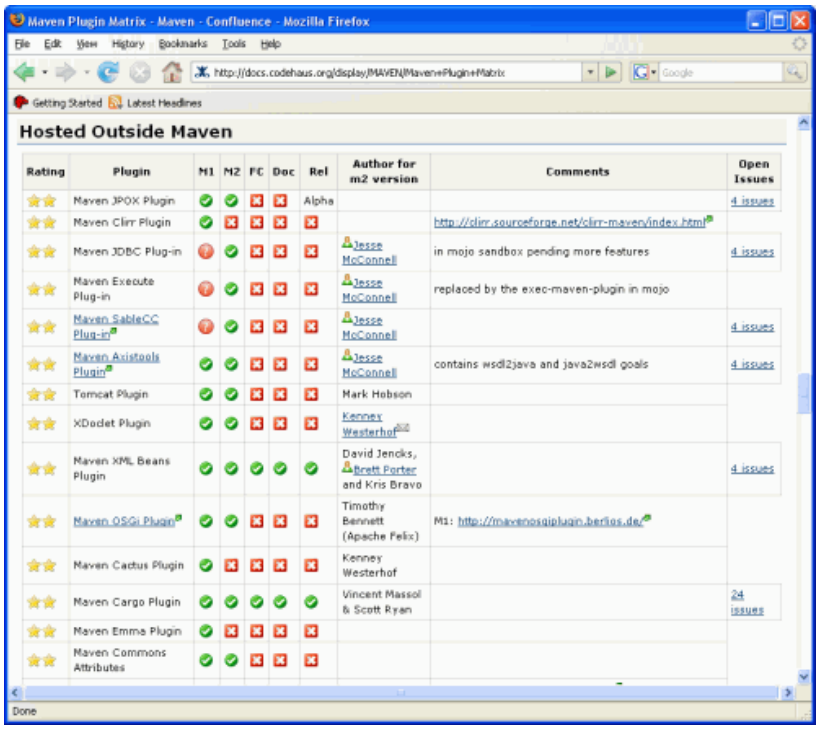
Mojos in plug-ins

Plug-ins are software modules written to fit into Maven's plug-in framework. Currently, custom plug-ins can be created using Java, Ant, or Beanshell. Each task within a plug-in is called a *mojo*. Sometimes, plug-ins are viewed as a set of related mojos. Creating custom Maven 2 plug-ins is beyond this tutorial's scope; see Related topics for more information.

Maven 2 comes prepackaged to download and work with many frequently used plug-ins. Most typical development tasks do not require the use of additional plug-ins.

Before you set out to write your own plug-ins, you should first consult the popular Maven 2 plug-in listing Web sites (see Related topics) to see if the plug-in you need is already available. Figure 5 shows the Maven Plugin Matrix (see Related topics), which provides compatibility information for many available plug-ins:

Figure 5. Maven Plugin Matrix



Binding mojos to life cycle phases

A mojo (build task) within a plug-in is executed when the Maven engine executes the corresponding phase on the build life cycle. The association between a plug-in's mojo and a phase of the life cycle is called a *binding*. Plug-in developers can flexibly associate one or more life-cycle phases with a plug-in.

Phases of the default life cycle

Maven's built-in understanding of a build life cycle consists of many distinct phases. Table 1 provides a brief description of each phase:

Table 1. Maven 2 default life-cycle phases

Life-cycle phase	Description
validate	Ensures that the current configuration and the content of the POM is valid. This includes validation of the tree of pom.xml files.
initialize	A chance to carry out any initialization prior to the main tasks in a build cycle.
generate-sources	A chance for code generators to start generating source code that can be processed or compiled in the later phases.
process-sources	Provided for the parsing, modification, and transformation of the source. Both regular and generated source code can be processed here.
generate-resources	A chance to generate non-source-code resources. This typically includes metadata files and configuration files.

process-resources	Handles the processing of the non-source-code resources. Modifications, transformation, and relocation of resources can occur during this phase.
compile	Compiles the source code. The compiled classes are placed into a target directory tree.
process-classes	Handles any class file transformation and enhancement steps. Bytecode weavers and instrumentation tools often operate during this phase.
generate-test-sources	A chance for mojos that generate unit-test code to operate.
process-test-sources	Executes any processing necessary on the test source code prior to compilation. Source code can be modified, transformed, or copied during this phase.
generate-test-resources	Allows for the generation of test-related (non-source-code) resources.
process-test-resources	Enables processing, transformation, and relocation of test-related resources.
test-compile	Compiles the source code of the unit tests.
test	Runs the compiled unit tests and tallies the results.
package	Bundles the executable binaries into a distribution archive, such as a JAR or WAR.
pre-integration-test	Prepares for integration testing. Integration testing in this case refers to testing of the code in (a controlled clone) of the actual deployment environment. This step can deploy the archive to a server for execution.
integration-test	Carries out actual integration tests.
post-integration-test	Unprepares for integration testing. This can involve reset or reinitialization of the testing environment.
verify	Verifies the validity and integrity of the deployable archive. After this phase, the archive will be installed.
install	Adds the archive to the local Maven directory. This makes it available for any other modules that may depend on it.
deploy	Adds the archive to a remote Maven directory. This can make the artifact available to a larger audience.

Maven captures more than a decade of project build management experience from the open source community. You will be hard-pressed to find a software project whose build cycle cannot fit into the life-cycle phases in Table 1.

When you start Maven 2's engine, it marches in order through each phase in Table 1 and executes any mojo that may be bound to that phase. Each mojo in turn can use Maven 2's rich POM support, dependency management, and access to build-state information in performing its dedicated task.

When you invoke the Maven 2 engine, you can specify a life-cycle phase as a command-line argument. The engine works through all the phases up to and including the specified phase. All mojos in the included phases are triggered.

This, in a nutshell, is how Maven 2 operates. You will see the operation first-hand in the next section. With a background understanding of Maven's operation, its dependency management model, and its POM, you'll find working hands-on with Maven 2 to be a straightforward exercise.

Downloading and installing Maven 2

Downloading and installing Maven 2 distills down to the following steps:

1. Download the Maven 2 binaries from the official Maven project site (see Related topics).
2. Unarchive the distribution binaries into a directory of your choice.
3. Add the *InstallationDirectory*\bin directory to your `PATH` variable.

To verify your installation, key the `mvn -help` command. You'll see the help page shown in Listing 3:

Listing 3. Using the mvn -help command to verify installation

```
C:\>mvn -help

usage: mvn [options] [<goal(s)>] [<phase(s)>]

Options:
 -C,--strict-checksums         Fail the build if checksums don't match
 -c,--lax-checksums            Warn if checksums don't match
 -P,--activate-profiles        Comma-delimited list of profiles to
                               activate
 -ff,--fail-fast               Stop at first failure in reactorized builds
 -fae,--fail-at-end            Only fail the build afterwards; allow all
                               non-impacted builds to continue
 -B,--batch-mode               Run in non-interactive (batch) mode
 -fn,--fail-never              NEVER fail the build, regardless of project
                               result
 -up,--update-plugins          Synonym for cpu
 -N,--non-recursive            Do not recurse into sub-projects
 -npr,--no-plugin-registry     Don't use ~/.m2/plugin-registry.xml for
                               plugin versions
 -U,--update-snapshots         Update all snapshots regardless of
                               repository policies
 -cpu,--check-plugin-updates   Force upToDate check for any relevant
                               registered plugins
 -npu,--no-plugin-updates      Suppress upToDate check for any relevant
                               registered plugins
 -D,--define                   Define a system property
 -X,--debug                    Produce execution debug output
 -e,--errors                   Produce execution error messages
 -f,--file                     Force the use of an alternate POM file.
 -h,--help                     Display help information
 -o,--offline                  Work offline
 -r,--reactor                  Execute goals for project found in the
                               reactor
 -s,--settings                 Alternate path for the user settings file
 -v,--version                  Display version information
```

Hands-on Maven 2 : Your first Maven 2 project

In the first hands-on example, you'll see how you can build simple projects using Maven 2 with minimal effort. Maven 2's built-in knowledge about Java projects eliminates tedious configuration that may be necessary with other build tools.

A class handling numeric operations

The example uses a class that handles numeric operations. The source code for the main class, called `NumOps`, is shown in Listing 4:

Listing 4. The NumOps class

```
package com.ibm.devworks;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class NumOps {
  private List <Operation> ops = new ArrayList
<Operation>();
  public  NumOps() {
    ops.add( new AddOps());
  }
  public Operation getOp(int i)
   { Operation retval;
   if (i > ops.size())
      { retval = null;
   } else {
     retval = (Operation) ops.get(i);
   } return retval;
  }
  public int size() {
    return ops.size();
  }
  public static void main( String[] args ) {
    NumOps nop = new NumOps();
    for (int i=0;  i < nop.size(); i++) {
      System.out.println( "2 " +
        nop.getOp(i).getDesc() +
        " 1 is " +
        nop.getOp(i).op(2,1) );
    }
  }
}
```

The `NumOps` class manages a set of objects capable of performing numeric operations on two integers. The main method creates a `NumOps` instance and then calls each of the objects managed by `NumOps`, calling its `getDesc()` method and `op()` method respectively. All of the objects managed by `NumOps` implement the `Operation` interface, defined in Operation.java and shown in Listing 5:

Listing 5. The Operation interface

```
package com.ibm.devworks;

public interface Operation {
   int op(int a, int b);
   String getDesc();
}
```

The only operation defined in this initial example is an `AddOps` class, shown in Listing 6:

Listing 6. The AddOps class

```
package com.ibm.devworks;

public class AddOps implements Operation {
   public int op(int a, int b) {
      return a+b;
   }
   public String getDesc() {
      return "plus";
   }
}
```

When you execute the `NumOps` class, it prints the following output:

```
 2 plus 1 is 3
```

Using Archetype to create the initial project

To create everything you need for a simple Java project that can be built using Maven, you can use the Archetype plug-in, which comes standard with Maven 2. Unlike the build-phase plug-ins, the Archetype plug-in runs outside of a Maven project build life-cycle and is used to *create* Maven projects. Issue the following command (type all of the command on one line) from the directory that you want to contain the `NumOps` project:

```
mvn archetype:create -DarchetypeGroupId=org.apache.maven.archetypes
  -DgroupId=com.ibm.devworks -DartifactId=NumOps
```

The command provides the Archetype plug-in with the coordinates of your module: `com.ibm.devworks/NumOps/1.0-SNAPSHOT`. You don't need to specify the version in this case because the Archetype plug-in always defaults to `1.0-SNAPSHOT`. This command creates a starter pom.xml file for the project, along with the conventional Maven 2 directory structure. You'll find the code in this tutorial's source-code download under the example1 directory (see Download).

The output should be similar to Listing 7:

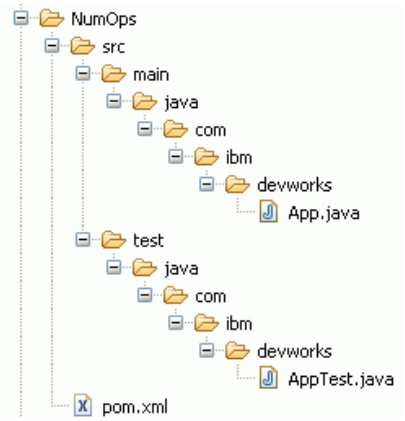
Listing 7. Using Maven Archetype to create the NumOps project

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
---
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:create] (aggregator-style)
[INFO] ------------------------------------------------------------------------
---
[INFO] Setting property: classpath.resource.loader.class => 'org.codehaus.plexus

...

[INFO] [archetype:create]
[INFO] Defaulting package to group ID: com.ibm.devworks
[INFO] ------------------------------------------------------------------------
---
[INFO] Using following parameters for creating Archetype: maven-archetype-quicks
tart:RELEASE
[INFO] ------------------------------------------------------------------------
---
[INFO] Parameter: groupId, Value: com.ibm.devworks
```

```
[INFO] Parameter: packageName, Value: com.ibm.devworks
[INFO] Parameter: basedir, Value: C:\temp\maven
[INFO] Parameter: package, Value: com.ibm.devworks
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: NumOps
[INFO] ********************* End of debug info from resources from generated POM
 **********************
[INFO] Archetype created in dir: C:\temp\maven\NumOps
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Sat Dec 02 22:04:02 EST 2006
[INFO] Final Memory: 4M/8M
[INFO] ------------------------------------------------------------------------
```

The Archetype plug-in creates a directory tree, a pom.xml file, and a placeholder App.java application. It also creates a directory tree for unit-test source code and a placeholder AppTest.java unit test. This project is ready to go. Figure 6 shows the directory and files created by the Archetype plug-in:

Figure 6. Archetype-generated directory and files



All you need to do is to move the NumOps.java, Operation.java, and AddOps.java files into the location where App.java is and remove App.java. In the next section, you'll make some changes to customize the generated pom.xml.

Customizing the POM

Maven 2 learns about your project via the pom.xml file. The file generated by the Archetype for `NumOps` is shown in Listing 8:

Listing 8. The Archetype-generated POM - 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>com.ibm.devworks</groupId>
  <artifactId>NumOps</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Note how the Archetype has defined the module's coordinates, defined the type as a JAR archive, and also specified JUnit as a dependency during the test phase (via the `<scope>` tag). To customize this pom.xml file for the new project, make the minor modifications highlighted in Listing 9:

Listing 9. Customizing the generated pom.xml for the NumOps project

```
<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>com.ibm.devworks</groupId>
  <artifactId>NumOps</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Intro to Maven 2 Example 1</name>  <url>http://www.ibm.com/java</url>   <build>    <plugins>
      <plugin>        <artifactId>maven-compiler-plugin</artifactId>        <configuration>          <source>1.5</
source>          <target>1.5</target>        </configuration>       </plugin>    </plugins>  </build>
       <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

The additional `<build>` tag is necessary to override the source and target the Java code level. By default, JDK 1.4 is assumed, but your code uses generics and requires JDK 5.0 compilation.

Compiling the customized project

You can now compile the `NumOps` project using the `mvn compile` command. This command causes the Maven 2 engine to march through the build life cycle to the compile phase, executing mojos along the way. You should see the report of a successful build, creating three class files in the

target tree (shown in Listing 10). This can take a little while if it is the first time you run it because some dependencies might need to be downloaded from the central repository over the Internet.

Listing 10. Output from mvn compile on the NumOps project

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
---
[INFO] Building Intro to Maven 2 Example 1
[INFO]    task-segment: [compile]
[INFO] ------------------------------------------------------------------------
---
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
Compiling 3 source files to C:\temp\maven\NumOps\target\classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Sat Dec 02 22:52:16 EST 2006
[INFO] Final Memory: 3M/7M
[INFO] ------------------------------------------------------------------------
```

Adding a unit test

Development best practices require unit tests on all code modules. Maven 2 created a placeholder AppTest.java unit test for you. Now rename the file to NumOpsTest.java and make the highlighted changes shown in Listing 11 to the generated unit test. You can also copy the unit test source code from the source code download (see Download).

Listing 11. Adding the NumOpsTest unit test to the project

```
package com.ibm.devworks;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
 * Unit test for simple App.
 */
public class NumOpsTest
    extends TestCase
{
    /**
     * Create the test case
     *
     * @param testName name of the test case
     */
    public NumOpsTest( String testName )
    {
        super( testName );
    }

    ...

    public void testNumOps()    {    NumOps nops = new NumOps();    assertTrue( nops.size() == 1);
        assertTrue(          nops.getOp(0).getDesc().equals("plus"));        assertTrue( nops.getOp(0).op(2,1) ==
 3);    }
}
```

You can now run all the mojos up to the test phase using the `mvn test` command.

Maven 2 compiles the source and the unit test. It then runs the tests, reporting on the number of successes, failures, and errors, as shown in Listing 12:

Listing 12. Executing mvn test to compile the project and run unit tests

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
---
[INFO] Building Intro to Maven 2 Example 1
[INFO]    task-segment: [test]
[INFO] ------------------------------------------------------------------------
---
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
Compiling 1 source file to C:\temp\maven\NumOps\target\test-classes
[INFO] [surefire:test]
[INFO] Surefire report directory: C:\temp\maven\NumOps\target\surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.ibm.devworks.NumOpsTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.031 sec

Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Sat Dec 02 23:04:27 EST 2006
[INFO] Final Memory: 3M/6M
[INFO] ------------------------------------------------------------------------
```

Hands-on Maven 2: Working with multiple project builds

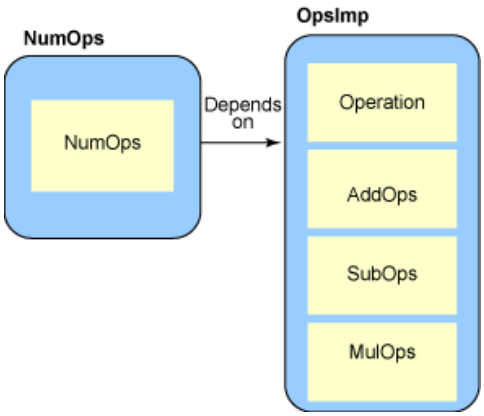
Building and testing simple projects using Maven 2 is straightforward. This section examines a second example demonstrating the more realistic and common case of a multiple-modules project.

Extending the NumOps example

The `NumOps` example is extended in this second example. A new `SubOps` class is added to support subtraction, and a new `MulOps` class is added to support multiplication.

However, the `Operation` interface and the `AddOps` class are now removed from the `NumOps` project. Instead, they are placed together with the new `SubOps` and `MulOps` classes in a new project called `OpsImp`. Figure 7 shows this relationship between the `NumOps` and `OpsImp` projects:

Figure 7. Relationship between NumOps and OpsImp



Dependencies among subprojects and submodules within a larger project is a frequently occurring scenario in software development. You can apply the technique shown here to any multimodule Maven project with interdependencies.

`SubOps`, shown in Listing 13, is coded similarly to `AddOps`. `MulOps`, not shown here, is similar; you can take a look at the code distribution for details (see Download).

Listing 13. The new SubOps class implementing the Operation interface

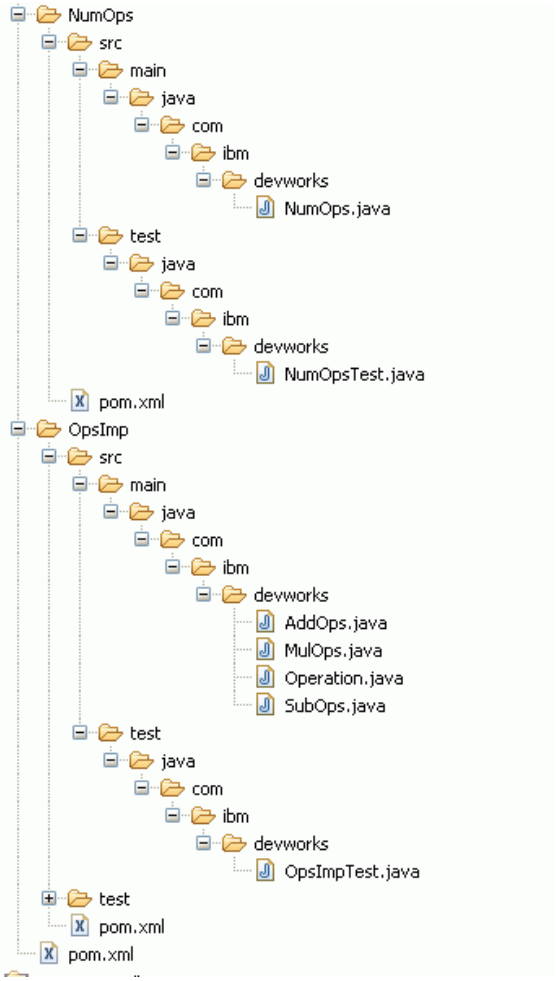
```
package com.ibm.devworks;

public class SubOps implements Operation {
   public int op(int a, int b) {
      return a-b;
   }
   public String getDesc() {
      return "minus";
   }
}
```

The constructor of `NumOps` has now been modified to create an instance of `SubOps` and an instance of `MulOps`. See the source code distribution for details.

Creating a master project

To work with these two projects, a master project has been created one directory above the `NumOps` and the `OpsImp` project directories. Both the `NumOps` and `OpsImp` projects use the standard Maven project directory layout. At the top level, the project directory consists of only a pom.xml file. Figure 8 shows the new sub-directory structure, immediately under the master directory:

Figure 8. Directory structure for a multimodule project



You can find the code for this multimodule project in the example2 subdirectory of the code distribution (see Download). The top-level pom.xml file is shown in Listing 14:

Listing 14. The top level pom.xml for the multimodule project

```
<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>com.ibm.devworks</groupId>
  <artifactId>mavenex2</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Example 2</name>
  <url>http://maven.apache.org</url>
<modules>
 <module>NumOps</module>
 <module>OpsImp</module>
</modules>
   <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
```

```
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
<dependencyManagement><dependencies>    <dependency>      <groupId>com.ibm.devworks</groupId>
      <artifactId>OpsImp</artifactId>      <version>${project.version}</version>    </dependency>  </
dependencies></dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

The new code is highlighted in bold. First, the artifact ID of this master project is `mavenex2`, and its packaging type is `pom`. This signals to Maven 2 that this is a multimodule project.

The `<modules>` tag then specifies the two modules that this project comprises: `NumOps` and `OpsImp`.

The submodules of this master project can inherit properties from this pom.xml file. More specifically, none of the submodules needs to declare JUnit as a dependency, even though they both contain unit tests. This is because they inherit the JUnit dependency defined at this top level.

The `<dependencyManagement>` tag does not specify dependencies that this module depends on. Instead, it is used mainly by submodules. Submodules can specify a dependency on any of the entries within the `<dependencyManagement>` tag without specifying a specific version number. This is useful for minimizing the number of edits required when a tree of projects changes dependency version numbers. In this case, the `OpsImp` project's version number is specified using `${project.version}` This is a parameter that will be filled with the appropriate value during Maven execution.

Inheriting from a master POM

Descending one level to the OpsImp directory, the pom.xml file for this module is shown in Listing 15:

Listing 15. The pom.xml file for the new OpsImp project

```
<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">
  <parent>
   <groupId>com.ibm.devworks</groupId>
    <artifactId>mavenex2</artifactId>
    <version>1.0-SNAPSHOT</version>
   </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>OpsImp</artifactId>
  <packaging>jar</packaging>
 </project>
```

The `<parent>` element specifies the master POM that this module inherits from. Inheriting from the parent module simplifies this pom.xml greatly. All that is necessary is to override the artifact ID and packaging. This module inherits the parent's dependency: the JUnit module.

The `NumOps` pom.xml also inherits from the parent and is also quite simple. This pom.xml is shown in Listing 16:

Listing 16. The pom.xml for the NumOps project showing POM inheritance

```
<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">
  <parent>
   <groupId>com.ibm.devworks</groupId>
    <artifactId>mavenex2</artifactId>
    <version>1.0-SNAPSHOT</version>
   </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>NumOps</artifactId>
  <packaging>jar</packaging>
  <dependencies>    <dependency>      <groupId>com.ibm.devworks</groupId>      <artifactId>OpsImp</
artifactId>     </dependency>  </dependencies>
</project>
```

Discovering the effective POM

Whenever you are inheriting from a higher-level POM, you can always find out what your equivalent pom.xml looks like after all the inherited elements have been accounted for. The command to show the "effective POM" is `mvn help:effective-pom`. You'll see some elements that you have not specified yourself. These are inherited from the Super POM. Every project pom.xml inherits implicitly from Maven's built-in Super POM.

The interesting item in the `NumOps` POM is the specification of the `OpsImp` project as a dependency. Note that no version number is specified in this dependency. The preferred version number is already specified within the parent's `<dependencyManagement>` element.

At the top-level project, you can now issue the `mvn compile` command to compile both modules or `mvn test` to run the unit tests of both modules. You can also run `mvn install` to install the packaged modules to your local directory. This allows any modules that depend on it to resolve the dependency without requiring access to the source code.

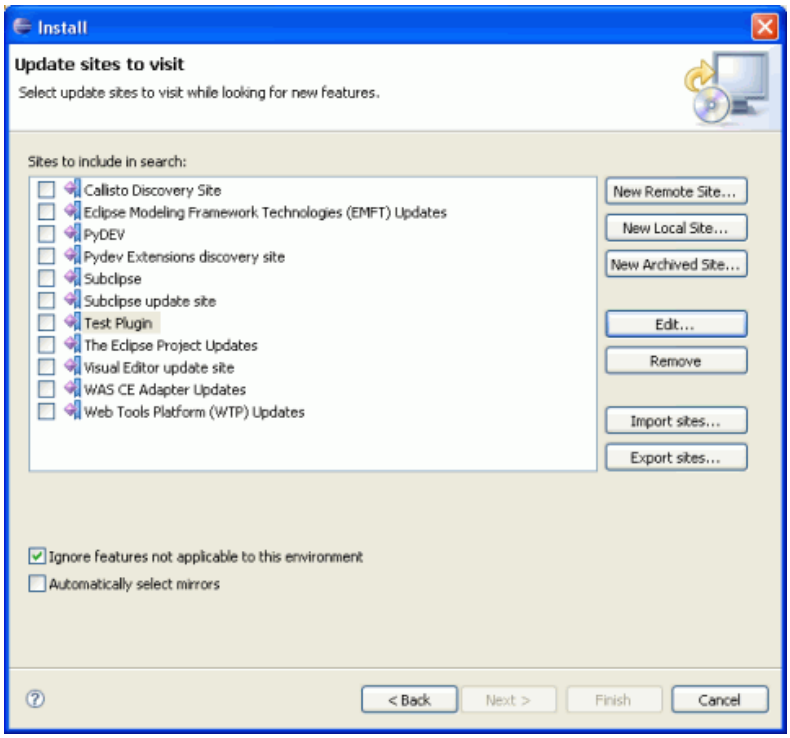
Installing the Maven 2.x Plug-in for Eclipse 3.2

If you use the Eclipse IDE for your daily development, you should download and install the Maven 2.x Plug-in for Eclipse. This plug-in facilitates your work with Maven projects within the IDE. See Related topics for project information on this plug-in.

You can use Eclipse's software update wizard to install the Maven 2.X Plug-in for Eclipse:

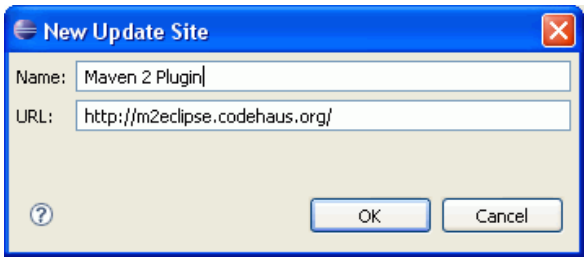
1. From Eclipse's **Help** menu, select **Software Updates>Find and Install...**.
2. Select **Search for new features to install** and click **Next**. You'll see the update sites wizard, shown in Figure 9:

Figure 9. Eclipse update sites wizard



3. Click the **New Remote Site** button.
4. In the pop-up dialog, enter the Maven 2.x Plug-in for Eclipse updates site, as shown in Figure 10:

Figure 10. Eclipse new remote site entry



5. Select the plug-in for installation and let the installation wizard restart the workspace. After the restart, you are ready to use the features of the Maven Plug-in for Eclipse.

Working with the Maven 2 Plug-in for Eclipse 3.2

This section covers some of the frequently used features of the Maven 2.x Plug-in for Eclipse.

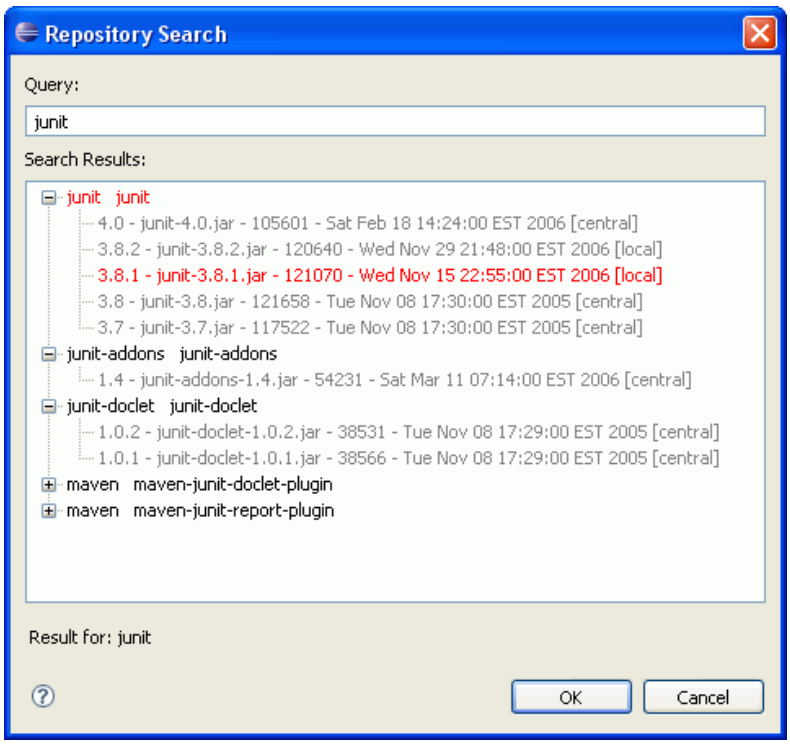
You need to enable Maven nature on an Eclipse project before the Maven 2 plug-in features are available with your project. Right-click on the project that you want to add Maven support to and select **Maven2>Enable**.

To ensure your project's directory structure reflects Maven's expectations, you should create your Maven directory structure (either manually or using an Archetype) first and then add the project to Eclipse.

Live repository search for dependencies

Adding dependencies to a pom.xml is easy using the plug-in. Right-click on the project's pom.xml and select **Maven2>Add Dependency**. This starts the Repository Search wizard. Type the first few characters of the name of the dependency you're looking for, and the wizard searches the central repository for any matching artifacts. All details of the matching artifacts are presented to you to help you select the dependency. Figure 11 shows the results of a search for JUnit artifacts:

Figure 11. Maven Repository Search wizard

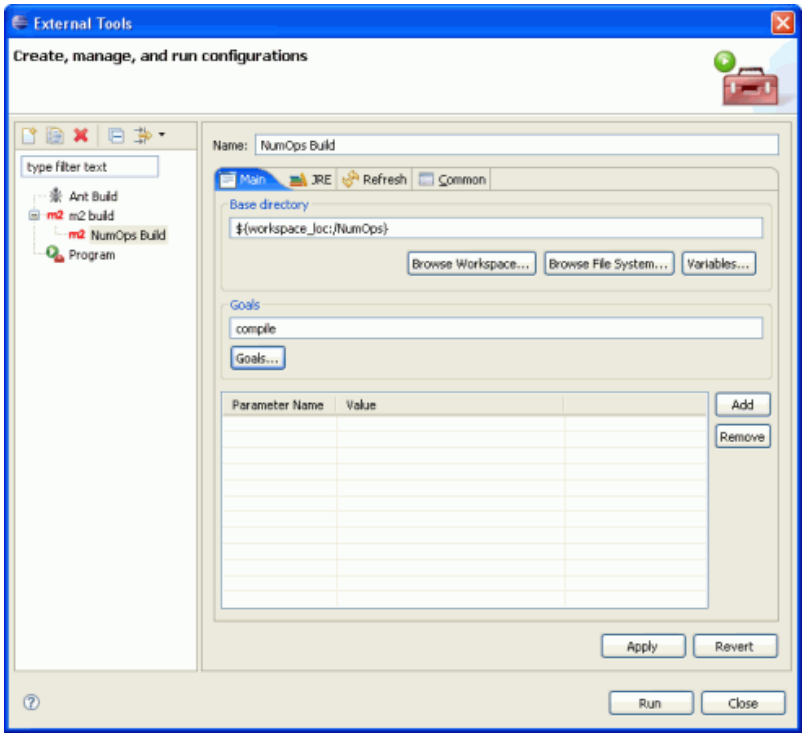


Once you have selected the artifact version you want and click **OK**, a new `<dependency>` element is added to the pom.xml by the plug-in automatically.

Invoking a Maven build

A build involving any of the life-cycle phases can be started from Eclipse. First, make sure the Maven-enabled project is currently open. Then, from the Eclipse menu, select **Run>External Tools>External Tools...**. This displays the External Tools wizard, as shown in Figure 12:

Figure 12. Building via Maven using the Eclipse External Tool wizard



Give this configuration a name and then select a life-cycle phase by clicking the **Goals...** button. Click **Run** to run Maven.

The Maven output is displayed on Eclipse's **Console** tab.

Summary

In this tutorial, you have:

- Explored the models and motivations behind Maven 2's design
- Gained an understanding of the all-important Maven POM
- Observed how the Maven repository and coordinate system simplifies complex dependency management
- Worked with Maven 2 for quick and simple projects
- Learned how Maven 2 can help in larger, multimodule projects
- Experimented with the Maven 2 Eclipse Plug-in
- Had a first-hand look at how Maven 2 can facilitate part of your daily project build and source/binary management activities

As software-development collaboration evolves, so will Maven evolve and adapt to its needs. As the backbone build facility for most large open source projects, it is guaranteed to benefit from continuous suggestions for improvement from the developer communities.

Learning Maven 2 need not be difficult, once you understand its motivation and the challenges that it aims to overcome. As a build tool, Maven 2 is usable productively even by rank beginners

developing projects in an isolated silo. Start using Maven 2 in your own development project or join one of the many open source development communities, and soon you too will be influencing the course toward which Maven 2 will evolve.

Downloadable resources

Description	Name	Size
Sample code for this tutorial	j-mavenv2.zip	34.1KB

Related topics

- Apache Maven 2: Download Maven 2 from the project site.
- Eclipse: Download the Eclipse SDK.
- Maven project Web site: Get up-to-date detailed information on Maven 2, including the settings reference and details on how to create custom plug-ins.
- Maven 2.x Plug-in for Eclipse : Download the latest version of the plug-in.
- The Apache Maven plug-ins list: Need more Maven mojos? These sites hold entire catalogs of available Maven plug-ins.