Mobiliser Platform – Foundation

Introduction to Maven

Maven

What it is and Why it is Needed?

Maven – Background to the Name

ma • ven [**mey**-*vuh*n]
noun
An expert or connoisseur

Also, **ma • vin**

Origin:
From the Hebrew ***mevin*** (מבִּין) meaning “one who understands”

Entered English via Yiddish.

The first recorded English usage occurred in a 1950 article in the Toronto Jewish Standard and later popularised in the 1960s through its usage in radio adverts in the United States for the food product *Vita Herring*.

Maven – What is it?

What it is

Maven started life as a framework to manage the complex software build processes needed for the Jakarta Turbine project. However, it has now grown to the point where it can be thought of simply as a container framework for handling abstract tasks.

This means it can handle not only the tasks related to building Java projects, but also project management, website generation and aiding communication between members of a development team.

However, Maven is still primarily used in its original capacity as a build automation tool for Java based software.

Maven – What Problem does is solve?

What problem does it solve?

Java development is now a distributed activity involving contributions from many people, operating in separate teams in different geographic locations.

Building the applications created in a collaborative development environment is a complex process that requires its own controlling software. Maven fulfils this role.

Maven – What does it do?

Maven is based on the assumption that no matter what Java project you are working on, all Java projects require the same basic work flow of tasks to be performed when the project is built.

Thus, in contrast to other build tools such as Ant, Maven adopts the approach known as ***convention-over-configuration***.

Maven – What does it do?

Maven is based on the assumption that no matter what Java project you are working on, all Java projects require the same basic work flow of tasks to be performed when the project is built.

Thus, in contrast to other build tools such as Ant, Maven adopts the approach known as ***convention-over-configuration***.

Based on Maven's basic assumptions about the standard tasks that should be performed when a Java project is built, it will:

- Understand how the structure of your project fits in to its standard model of how all Java projects are built

Maven – What does it do?

Maven is based on the assumption that no matter what Java project you are working on, all Java projects require the same basic work flow of tasks to be performed when the project is built.

Thus, in contrast to other build tools such as Ant, Maven adopts the approach known as ***convention-over-configuration***.

Based on Maven's basic assumptions about the standard tasks that should be performed when a Java project is built, it will:

- Understand how the structure of your project fits in to its standard model of how all Java projects are built
- Follow a standardized sequence of build phases – each of which can be configured, modified, extended or simply skipped

Maven – What does it do?

Maven is based on the assumption that no matter what Java project you are working on, all Java projects require the same basic work flow of tasks to be performed when the project is built.

Thus, in contrast to other build tools such as Ant, Maven adopts the approach known as ***convention-over-configuration***.

Based on Maven's basic assumptions about the standard tasks that should be performed when a Java project is built, it will:

- Understand how the structure of your project fits in to its standard model of how all Java projects are built
- Follow a standardized sequence of build phases – each of which can be configured, modified, extended or simply skipped
- Propose sensible default values in the absence of user specified values

Maven – What does it do?

Maven is based on the assumption that no matter what Java project you are working on, all Java projects require the same basic work flow of tasks to be performed when the project is built.

Thus, in contrast to other build tools such as Ant, Maven adopts the approach known as ***convention-over-configuration***.

Based on Maven's basic assumptions about the standard tasks that should be performed when a Java project is built, it will:

- Understand how the structure of your project fits in to its standard model of how all Java projects are built
- Follow a standardized sequence of build phases – each of which can be configured, modified, extended or simply skipped
- Propose sensible default values in the absence of user specified values
- Manage the resolution of software dependencies that exist within your project

Maven – What does it do?

Maven is based on the assumption that no matter what Java project you are working on, all Java projects require the same basic work flow of tasks to be performed when the project is built.

Thus, in contrast to other build tools such as Ant, Maven adopts the approach known as ***convention-over-configuration***.

Based on Maven's basic assumptions about the standard tasks that should be performed when a Java project is built, it will:

- Understand how the structure of your project fits in to its standard model of how all Java projects are built
- Follow a standardized sequence of build phases – each of which can be configured, modified, extended or simply skipped
- Propose sensible default values in the absence of user specified values
- Manage the resolution of software dependencies that exist within your project
- Provide a high degree of flexibility for each step of the build process

Maven – What does it do?

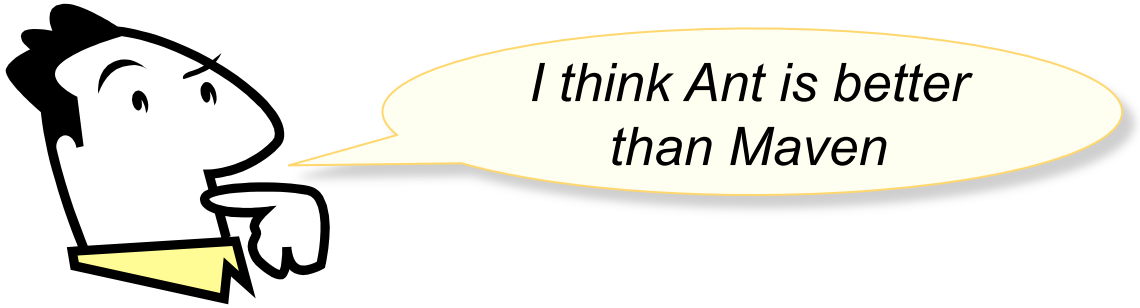
Maven is based on the assumption that no matter what Java project you are working on, all Java projects require the same basic work flow of tasks to be performed when the project is built.

Thus, in contrast to other build tools such as Ant, Maven adopts the approach known as ***convention-over-configuration***.

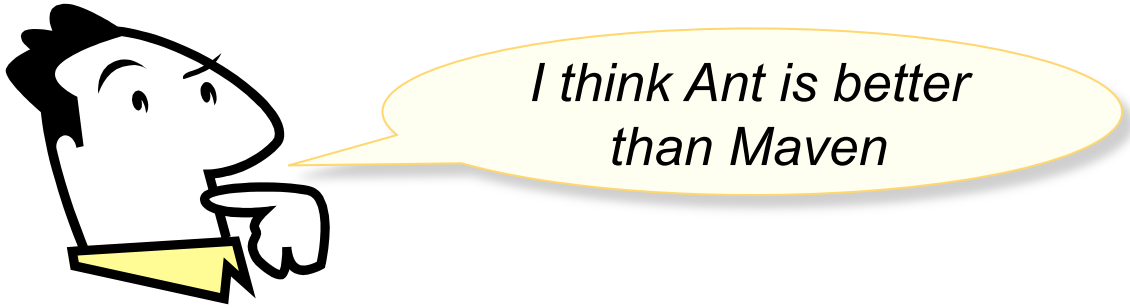
Based on Maven's basic assumptions about the standard tasks that should be performed when a Java project is built, it will:

- Understand how the structure of your project fits in to its standard model of how all Java projects are built
- Follow a standardized sequence of build phases – each of which can be configured, modified, extended or simply skipped
- Propose sensible default values in the absence of user specified values
- Manage the resolution of software dependencies that exist within your project
- Provide a high degree of flexibility for each step of the build process
- Allow a wide variety of new functionality to be added to the standard build process by means of plug-ins

Maven and Other Build Tools



Maven and Other Build Tools



Maven and Other Build Tools

Discussions such as these have largely missed the point for the same reason that the statement "*apples are better than oranges"* misses the point.

Maven and Other Build Tools

Discussions such as these have largely missed the point for the same reason that the statement "*apples are better than oranges"* misses the point.

Ant and Make are both highly flexible ***build tools***, but these tools make no assumptions about how your project should be built. You must supply ***all*** the necessary configuration before they will work.

Maven and Other Build Tools

Discussions such as these have largely missed the point for the same reason that the statement "*apples are better than oranges"* misses the point.

Ant and Make are both highly flexible ***build tools***, but these tools make no assumptions about how your project should be built. You must supply ***all*** the necessary configuration before they will work.

Maven, on the other hand, is entire ***build framework*** that comes with a pre-configured set of build tasks common to all Java projects.

Maven and Other Build Tools

Discussions such as these have largely missed the point for the same reason that the statement "*apples are better than oranges"* misses the point.

Ant and Make are both highly flexible ***build tools***, but these tools make no assumptions about how your project should be built. You must supply ***all*** the necessary configuration before they will work.

Maven, on the other hand, is entire ***build framework*** that comes with a pre-configured set of build tasks common to all Java projects.

Ant build steps are frequently incorporated into Maven build phases.

Maven

A Conceptual Overview

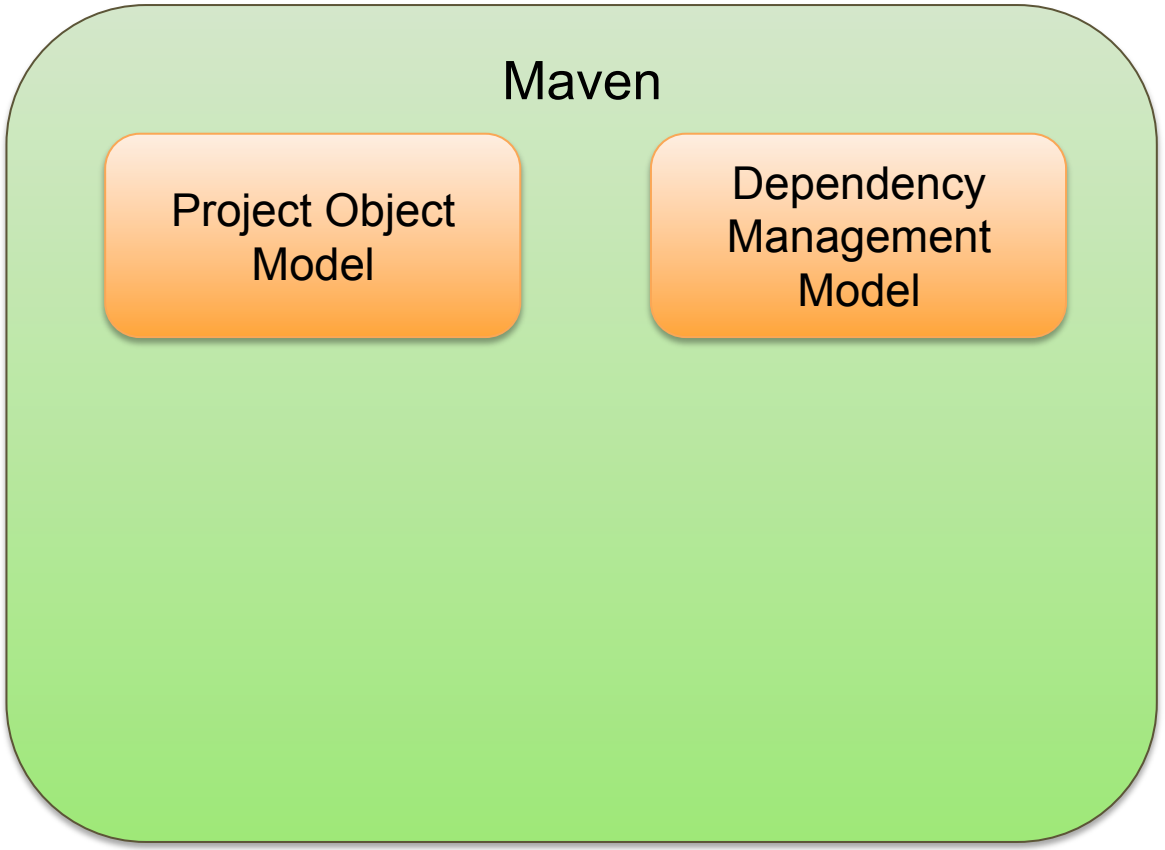
Maven: Conceptual Overview 1/4

Maven uses a model based approach for understanding:

Maven: Conceptual Overview 1/4

Maven uses a model based approach for understanding:

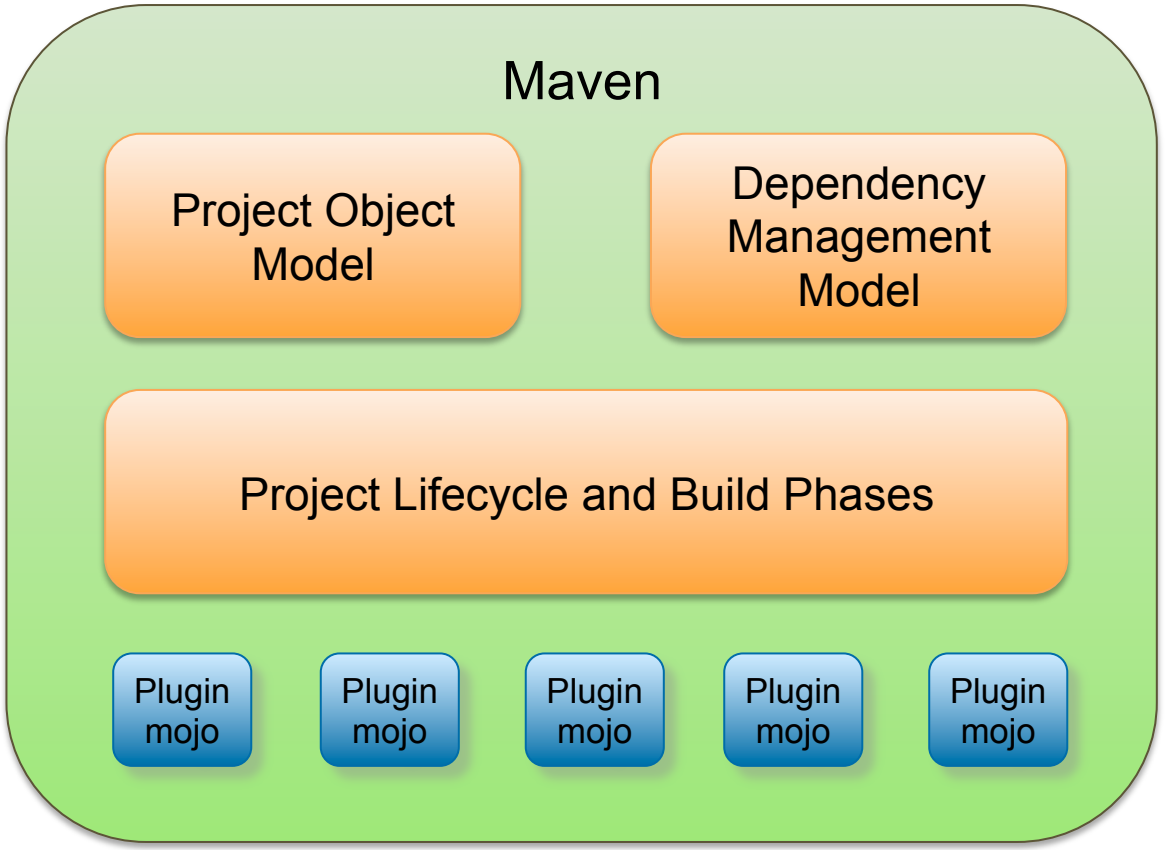
1. The structure of your project – what software units it consists of and how they are related



Maven: Conceptual Overview 1/4

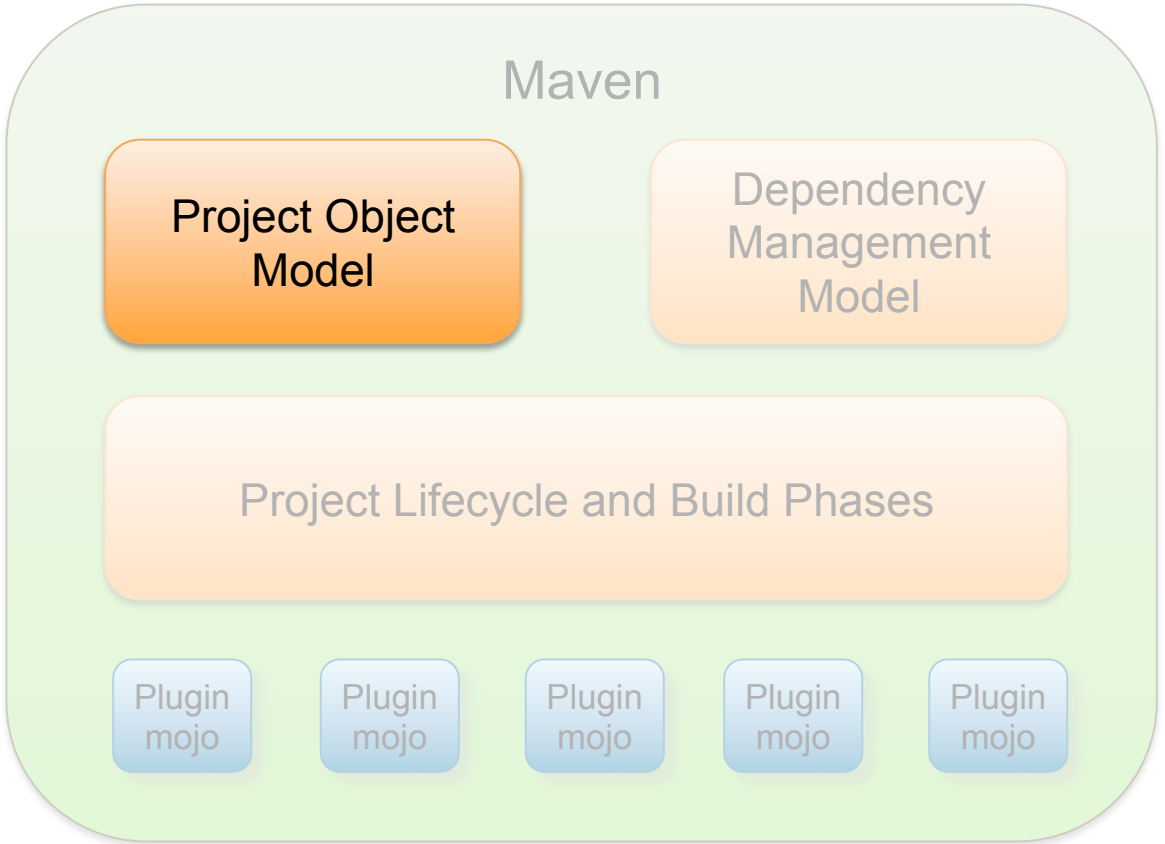
Maven uses a model based approach for understanding:

1. The structure of your project – what software units it consists of and how they are related
2. Which of its standardised build steps are needed to build your project



Maven: Conceptual Overview 2/4

The central model for controlling the behaviour of Maven is the ***Project Object Model***.

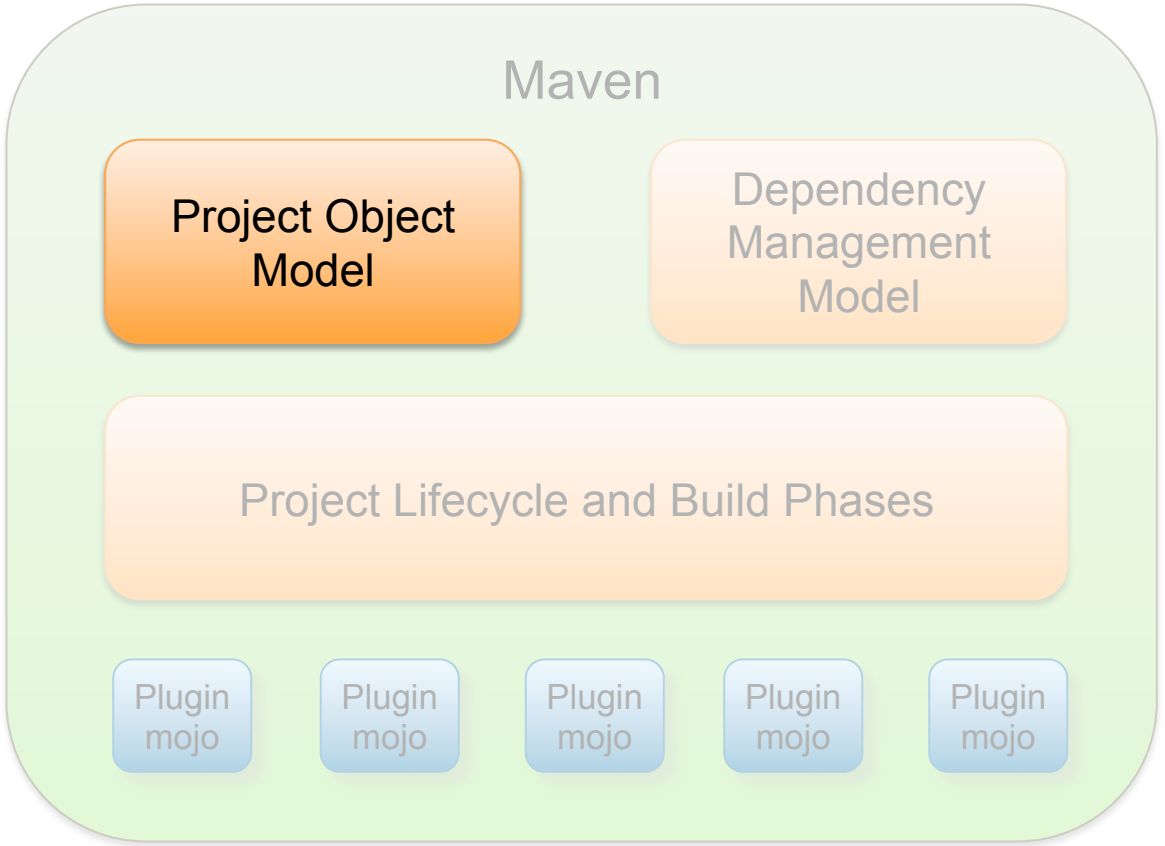


Maven: Conceptual Overview 2/4

The central model for controlling the behaviour of Maven is the ***Project Object Model***.

The Project Object Model (or POM) is composed of two distinct parts:

1. A hard-coded set of build steps built-in to Maven's core functionality

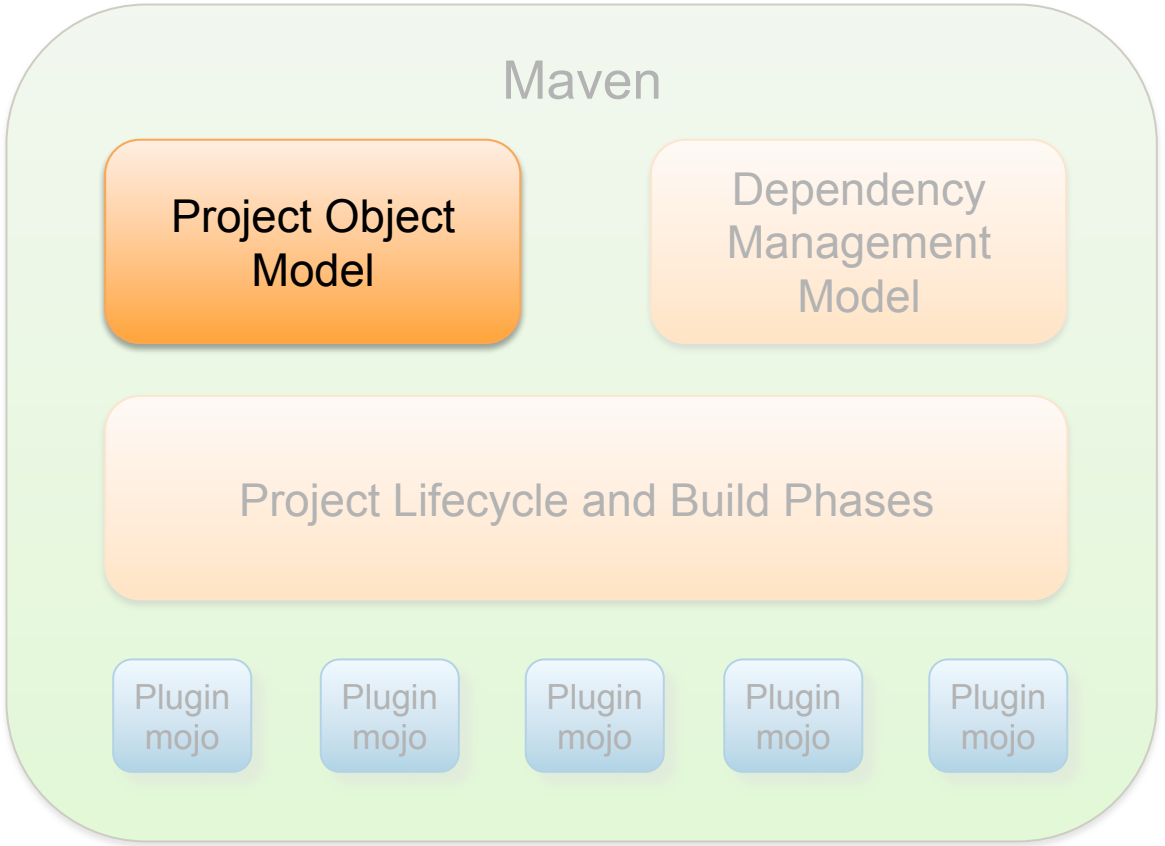


Maven: Conceptual Overview 2/4

The central model for controlling the behaviour of Maven is the ***Project Object Model***.

The Project Object Model (or POM) is composed of two distinct parts:

1. A hard-coded set of build steps built-in to Maven's core functionality
2. A user-defined set of declarations that describe the structure of your particular project including its dependencies.



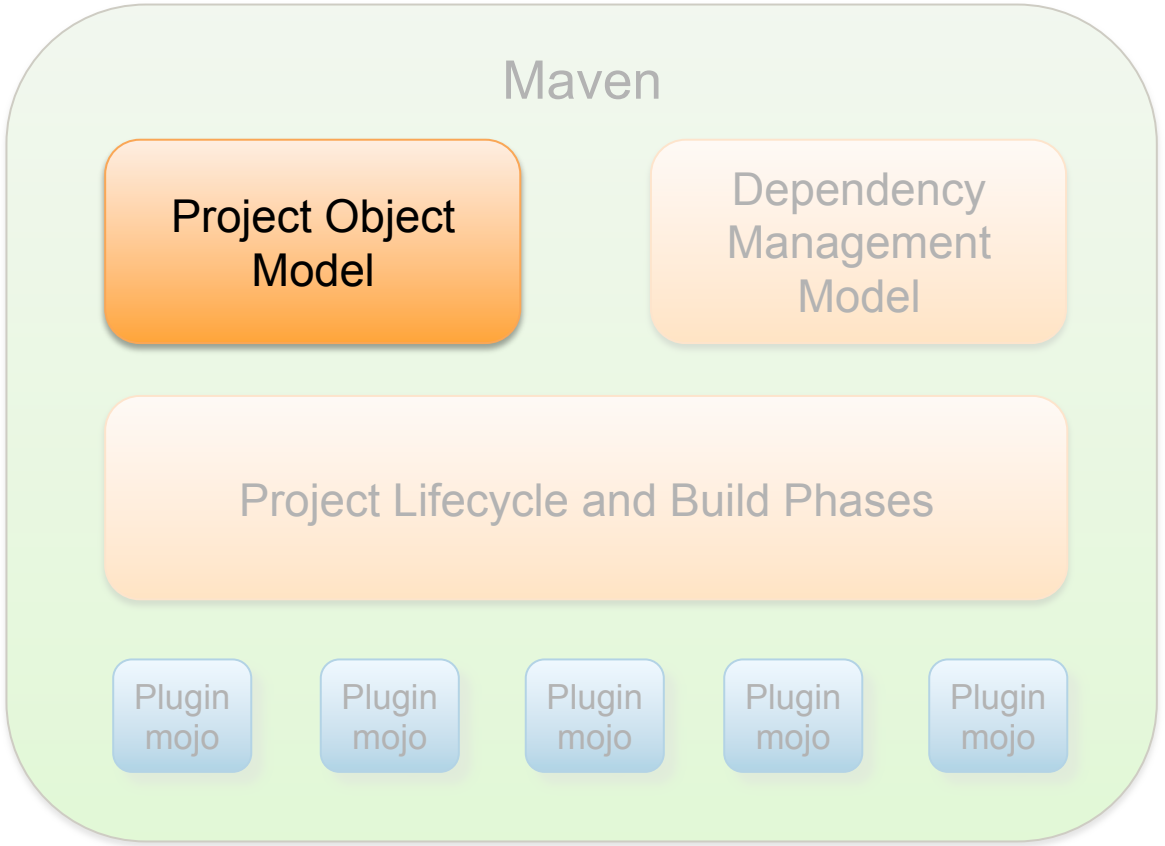
Maven: Conceptual Overview 2/4

The central model for controlling the behaviour of Maven is the ***Project Object Model***.

The Project Object Model (or POM) is composed of two distinct parts:

1. A hard-coded set of build steps built-in to Maven's core functionality
2. A user-defined set of declarations that describe the structure of your particular project including its dependencies.

The user-defined part of the POM is an XML file (`pom.xml`) that contains all the configuration necessary to describe a single project.



Maven: Conceptual Overview 2/4

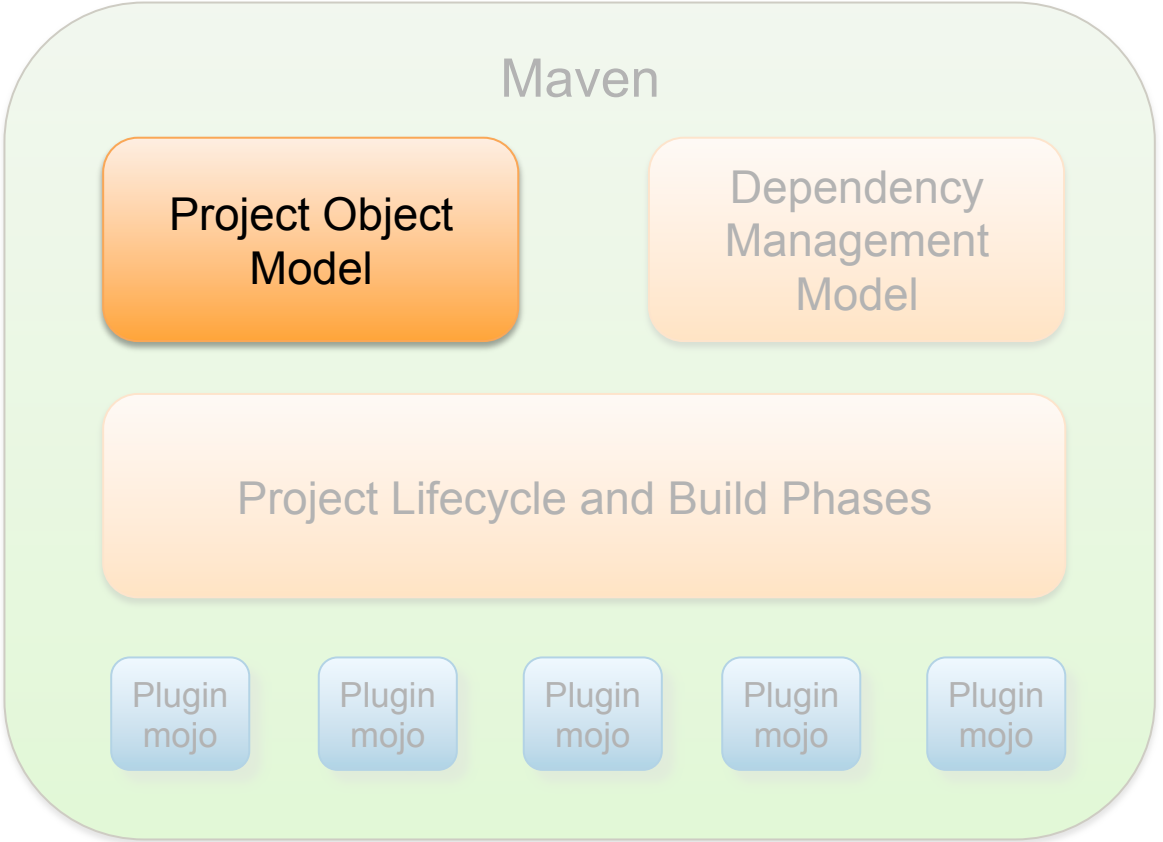
The central model for controlling the behaviour of Maven is the ***Project Object Model***.

The Project Object Model (or POM) is composed of two distinct parts:

1. A hard-coded set of build steps built-in to Maven's core functionality
2. A user-defined set of declarations that describe the structure of your particular project including its dependencies.

The user-defined part of the POM is an XML file (`pom.xml`) that contains all the configuration necessary to describe a single project.

Large projects can be divided into sub-projects, each with their own `pom.xml` file.



Maven: Conceptual Overview 2/4

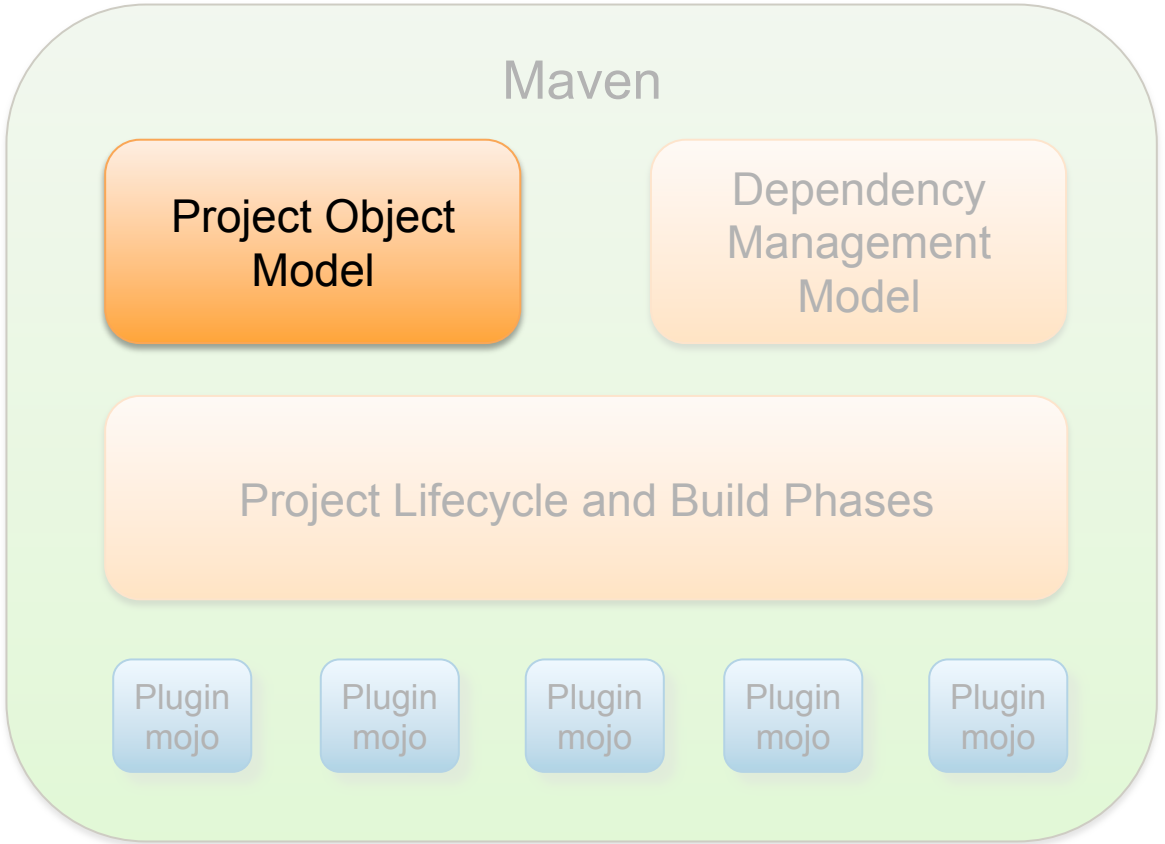
The central model for controlling the behaviour of Maven is the ***Project Object Model***.

The Project Object Model (or POM) is composed of two distinct parts:

1. A hard-coded set of build steps built-in to Maven's core functionality
2. A user-defined set of declarations that describe the structure of your particular project including its dependencies.

The user-defined part of the POM is an XML file (`pom.xml`) that contains all the configuration necessary to describe a single project.

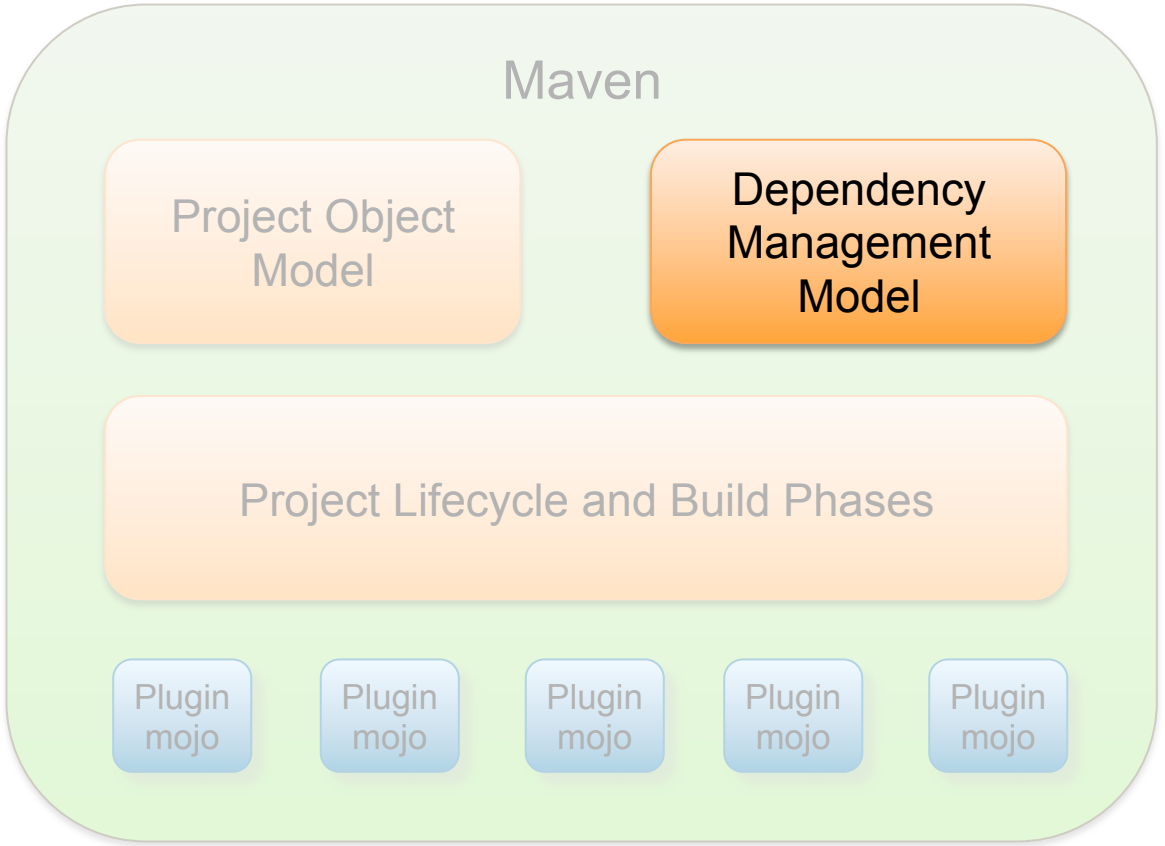
Large projects can be divided into sub-projects, each with their own `pom.xml` file.



The build process for any particular project is controlled by a root `pom.xml` file that contains references to the sub-project `pom.xml` files.

Maven: Conceptual Overview 3/4

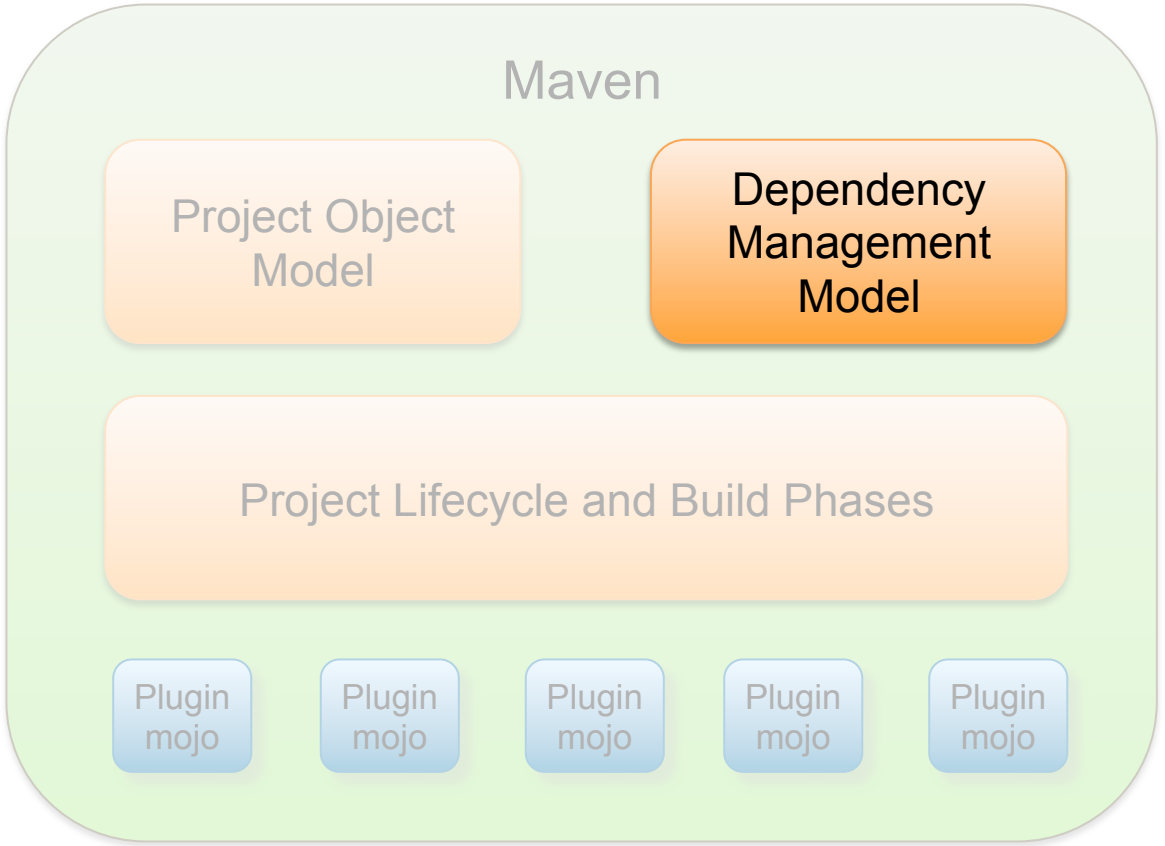
The `pom.xml` file contains references to all the units of software upon which your project depends.



Maven: Conceptual Overview 3/4

The `pom.xml` file contains references to all the units of software upon which your project depends.

In Maven terminology, any unit of software upon which your project depends is known as an ***artifact***.

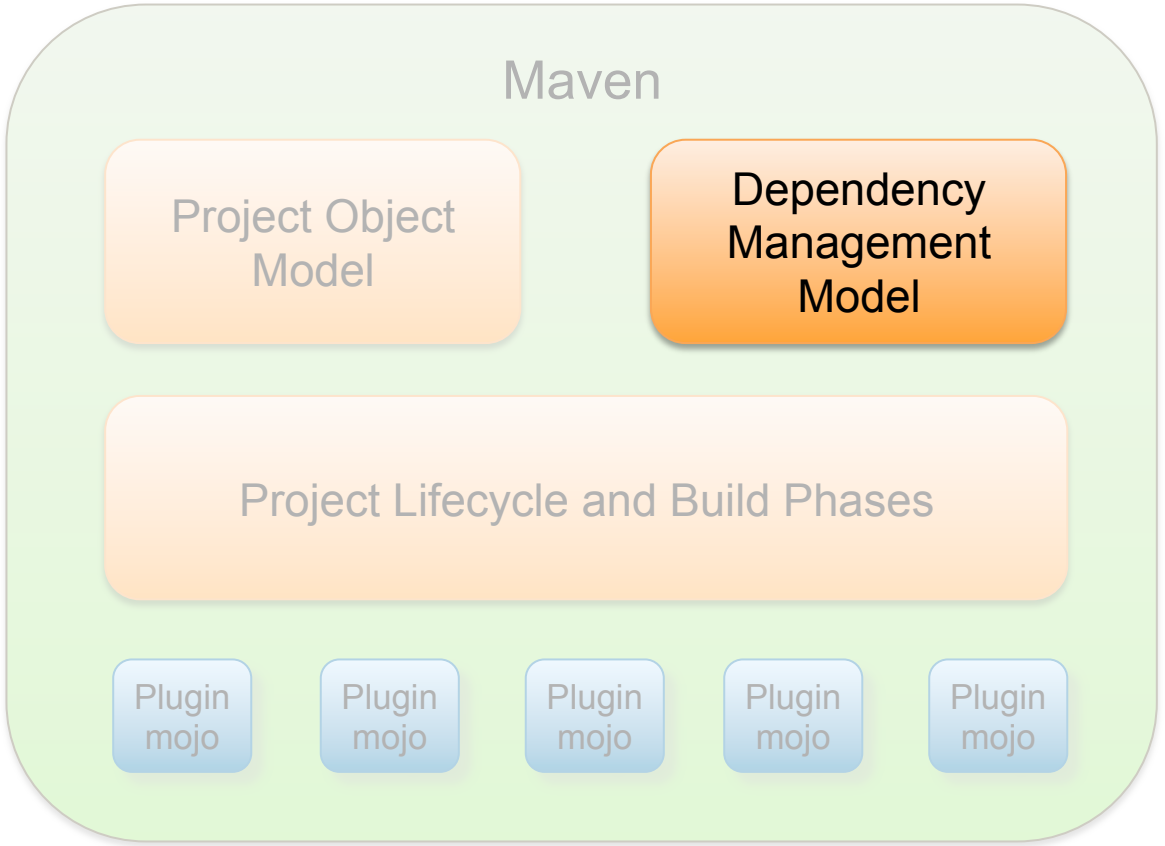


Maven: Conceptual Overview 3/4

The `pom.xml` file contains references to all the units of software upon which your project depends.

In Maven terminology, any unit of software upon which your project depends is known as an ***artifact***.

It is entirely possible that the artifacts needed for dependency resolution are not stored on your local machine.



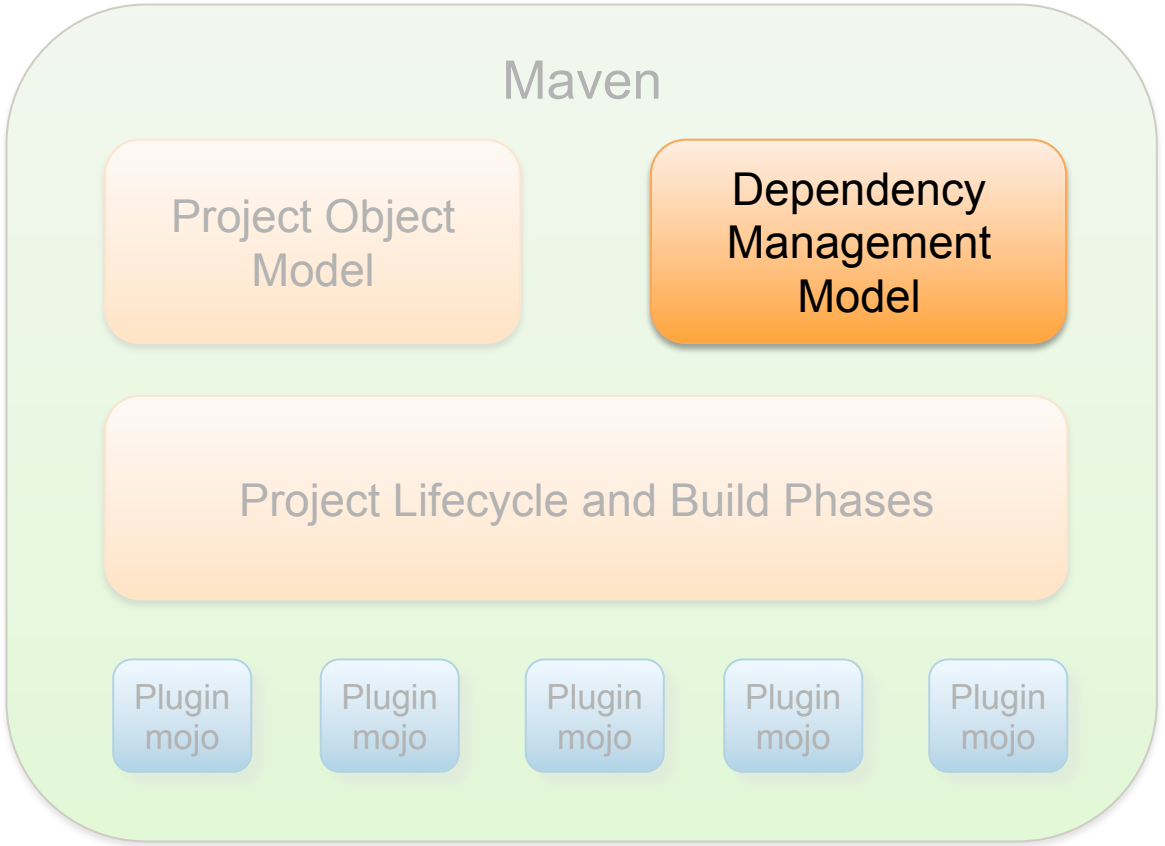
Maven: Conceptual Overview 3/4

The `pom.xml` file contains references to all the units of software upon which your project depends.

In Maven terminology, any unit of software upon which your project depends is known as an ***artifact***.

It is entirely possible that the artifacts needed for dependency resolution are not stored on your local machine.

In this case, the Dependency Management Model holds a list of the local and remote repositories from which the required artifacts can be obtained. This list can be found in the file `settings.xml`



Maven: Conceptual Overview 3/4

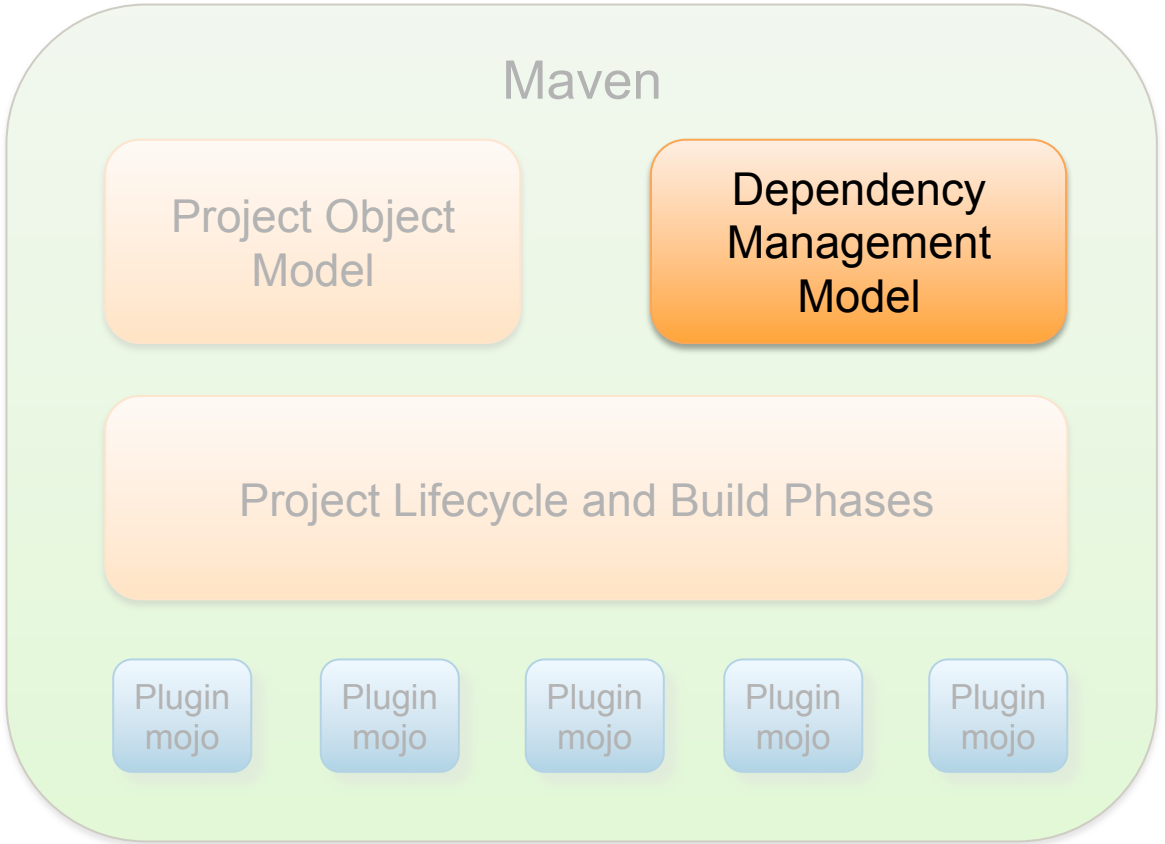
The `pom.xml` file contains references to all the units of software upon which your project depends.

In Maven terminology, any unit of software upon which your project depends is known as an ***artifact***.

It is entirely possible that the artifacts needed for dependency resolution are not stored on your local machine.

In this case, the Dependency Management Model holds a list of the local and remote repositories from which the required artifacts can be obtained. This list can be found in the file `settings.xml`

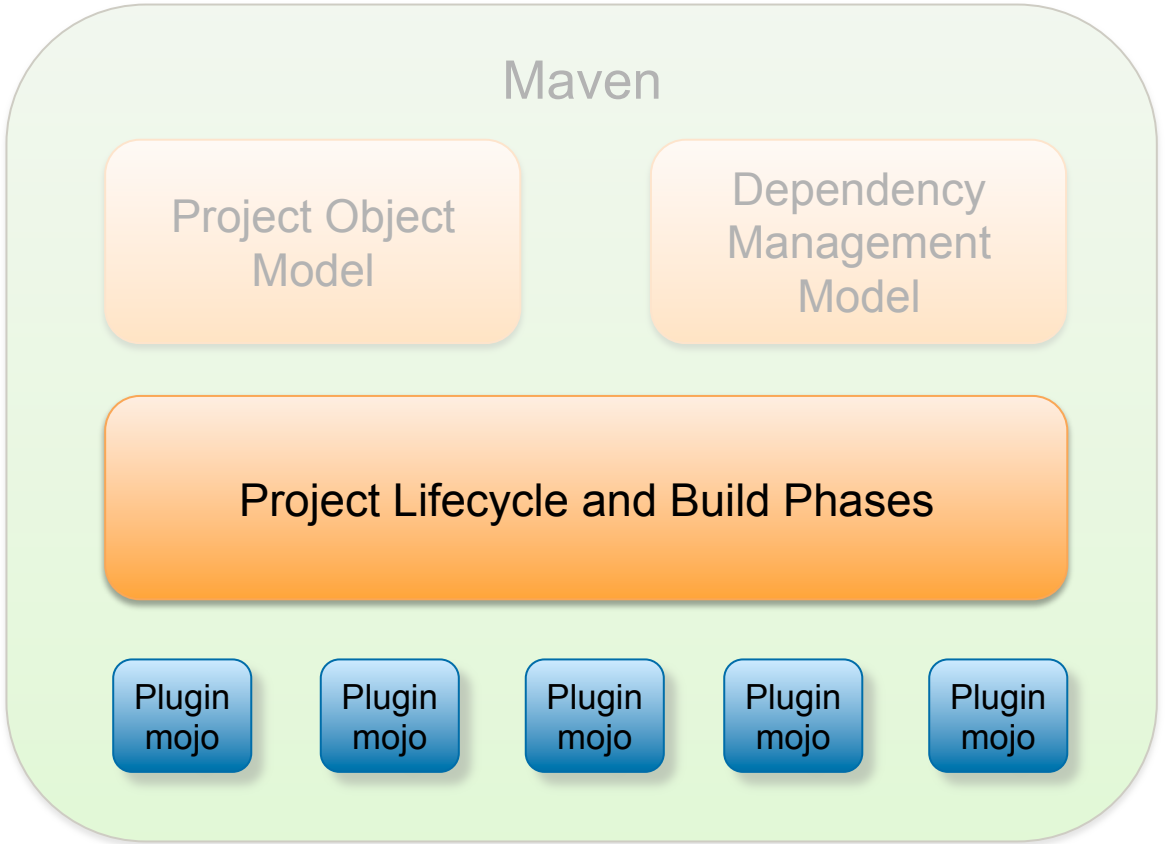
Any artifacts that are found to originate in a remote repository will be downloaded and held locally in your `~/.m2/repository` directory



Maven: Conceptual Overview 4/4

Every time Maven builds a project, it runs through a fixed sequence of build phases.

The functionality performed during each of these phases is implemented by means of plug-ins known as “Mojos”.

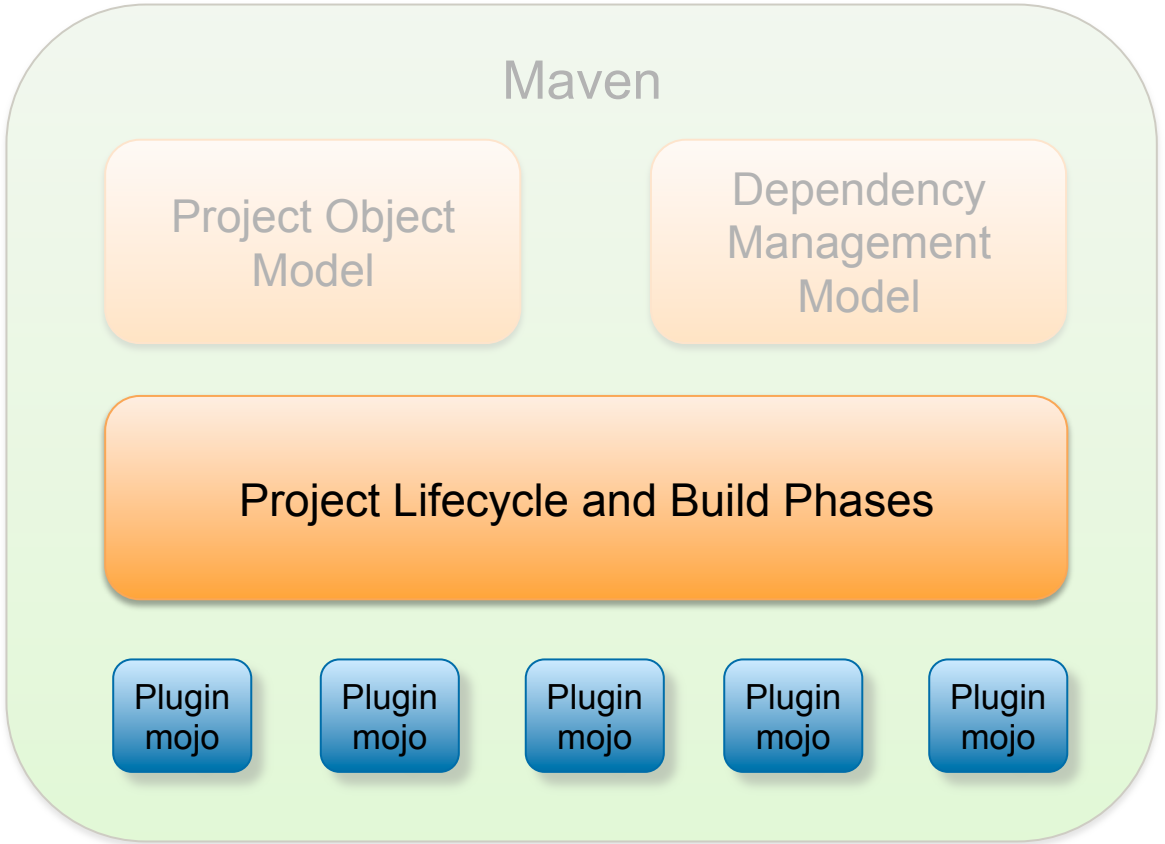


Maven: Conceptual Overview 4/4

Every time Maven builds a project, it runs through a fixed sequence of build phases.

The functionality performed during each of these phases is implemented by means of plug-ins known as “Mojos”.

Maven’s plug-in Mojos are themselves handled as if they were artifacts. The configuration for these Mojos is contained within a Super POM file from which all other project `pom.xml` files inherit their fundamental behaviour.



Maven

Standard Lifecycles

Maven Standard Lifecycles 1/2

Maven comes preconfigured with three standard lifecycles known as `default`, `clean` & `site`.

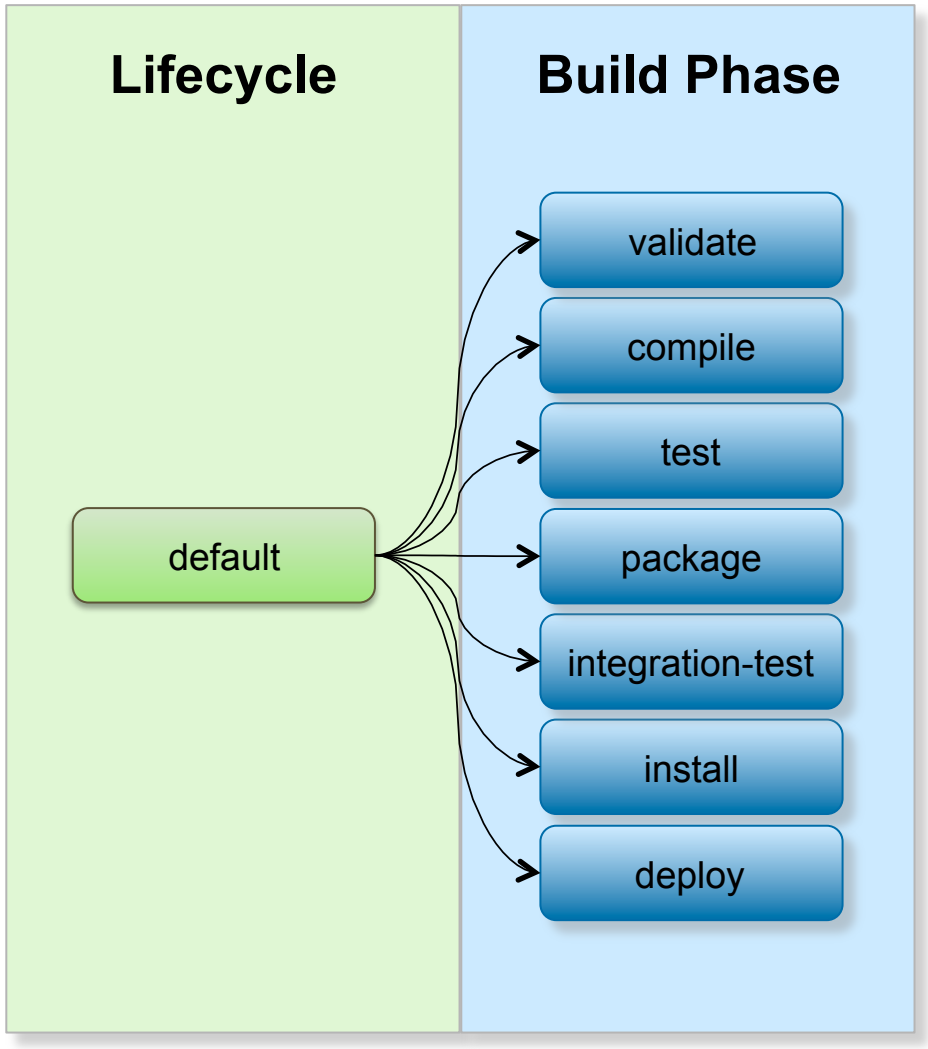
`default` is the most widely used of the Maven lifecycles and is shown here as it would be executed if you wished to create a jar file as the deployable output.

Maven Standard Lifecycles 1/2

Maven comes preconfigured with three standard lifecycles known as `default`, `clean` & `site`.

`default` is the most widely used of the Maven lifecycles and is shown here as it would be executed if you wished to create a jar file as the deployable output.

Each lifecycle is composed of a set of build phases that are executed in order.



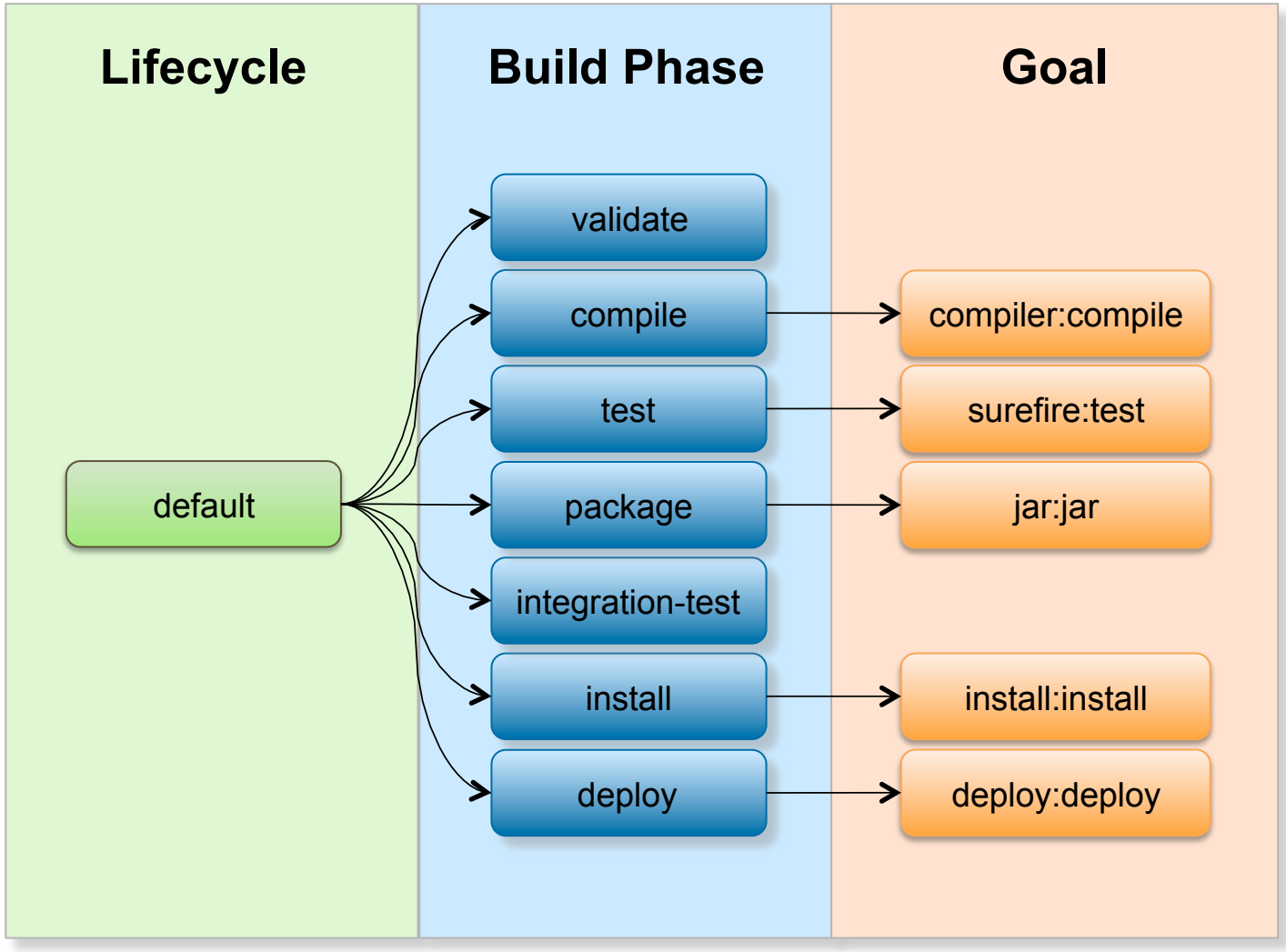
Maven Standard Lifecycles 1/2

Maven comes preconfigured with three standard lifecycles known as `default`, `clean` & `site`.

`default` is the most widely used of the Maven lifecycles and is shown here as it would be executed if you wished to create a jar file as the deployable output.

Each lifecycle is composed of a set of build phases that are executed in order.

Each build phase can have zero or more goals bound to it, where a goal is the actual functionality implemented as a plug-in Mojo.



Maven Standard Lifecycles 1/2

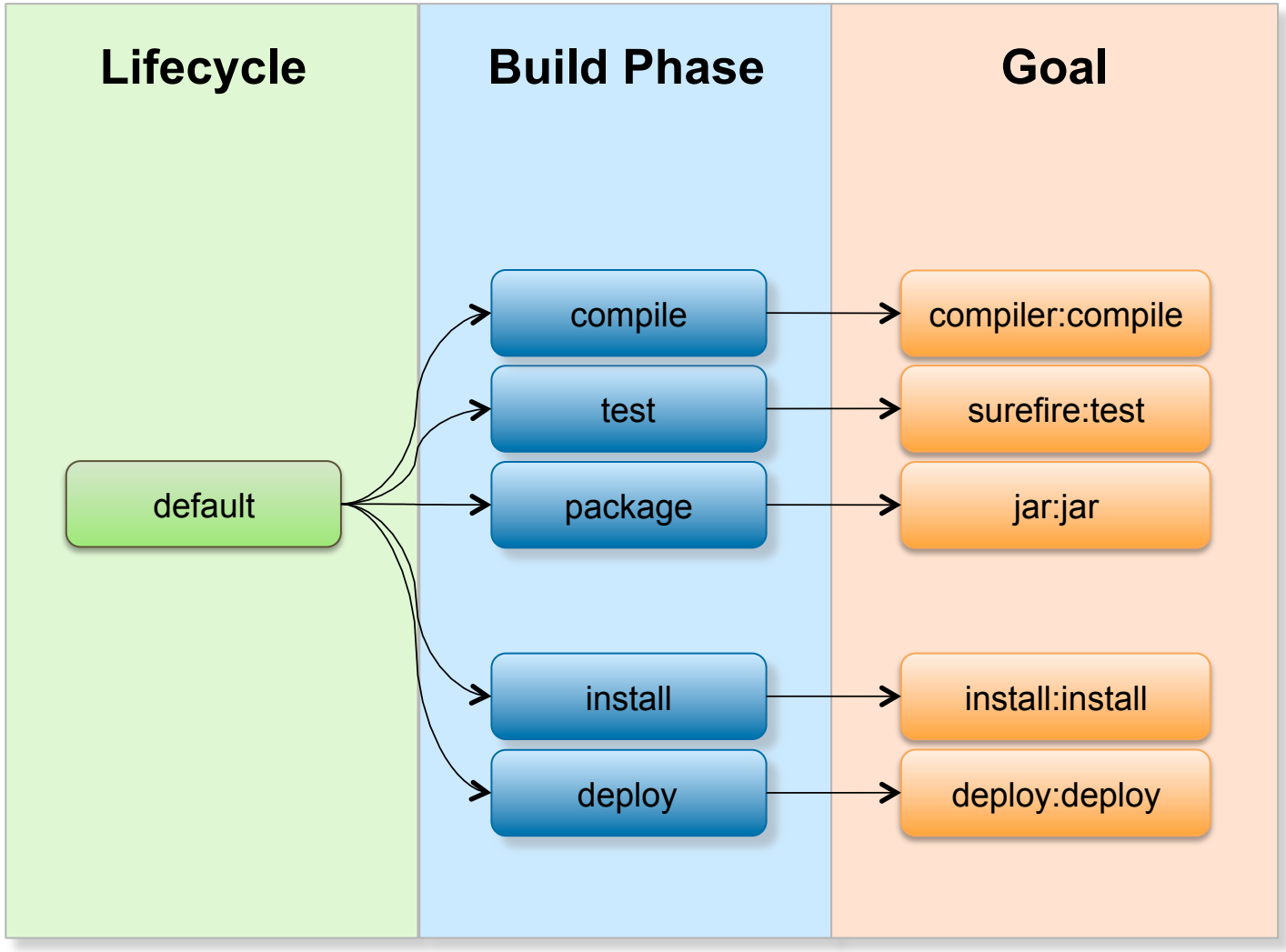
Maven comes preconfigured with three standard lifecycles known as `default`, `clean` & `site`.

`default` is the most widely used of the Maven lifecycles and is shown here as it would be executed if you wished to create a jar file as the deployable output.

Each lifecycle is composed of a set of build phases that are executed in order.

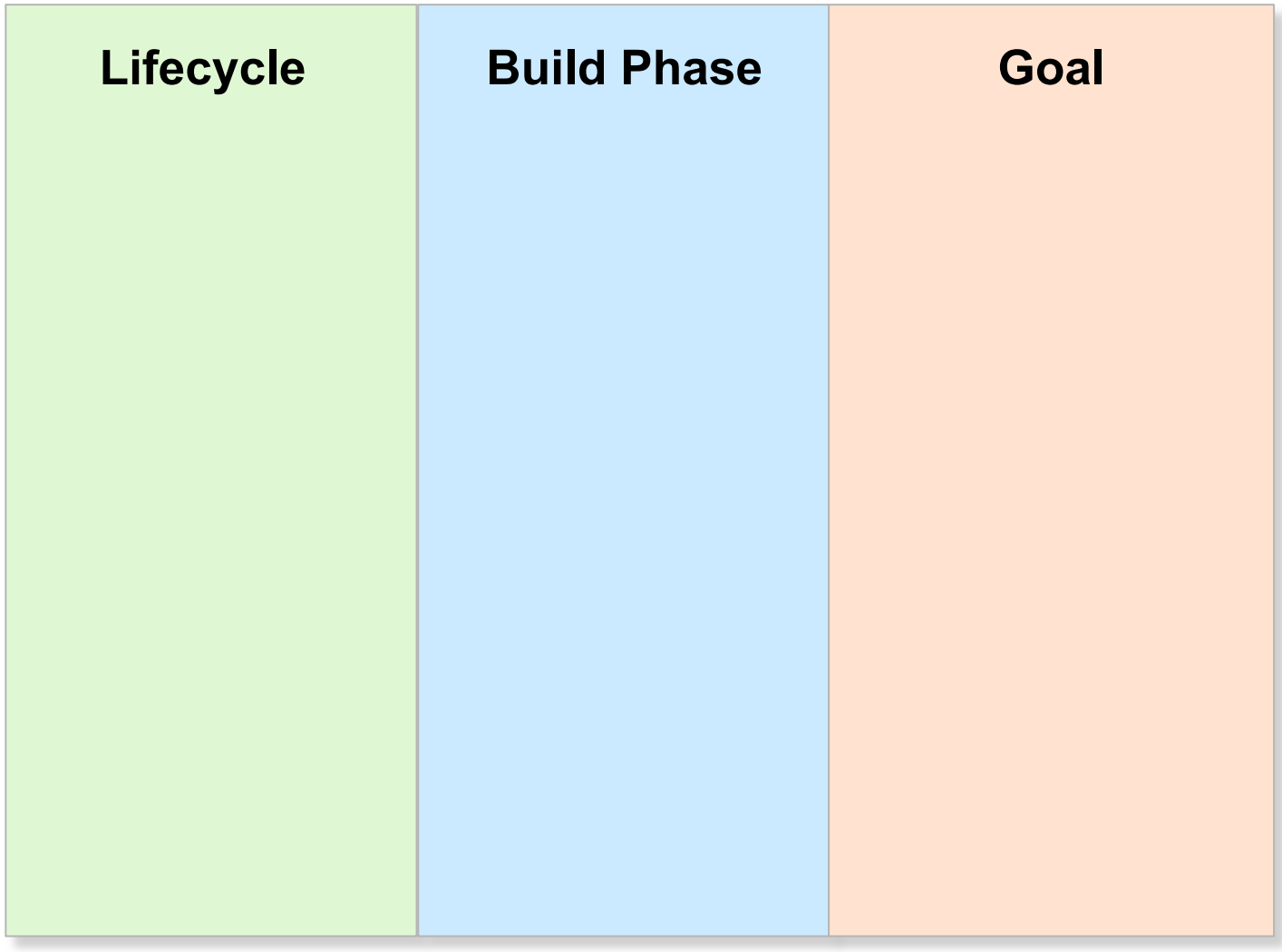
Each build phase can have zero or more goals bound to it, where a goal is the actual functionality implemented as a plug-in Mojo.

If no goals are bound to a build phase, then it is automatically skipped.



Maven Standard Lifecycles 2/2

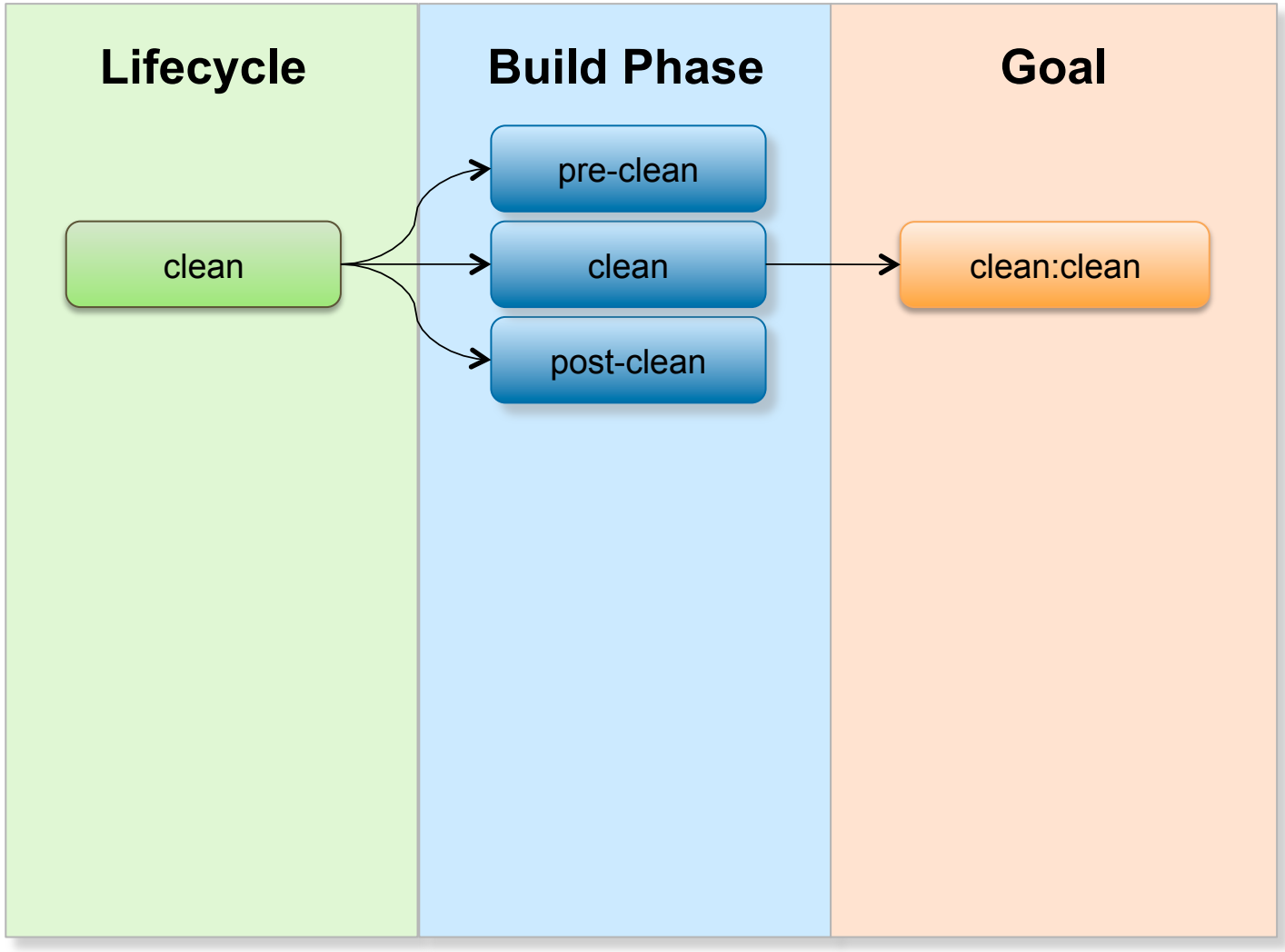
The other standard lifecycles are `clean` & `site`.



Maven Standard Lifecycles 2/2

The other standard lifecycles are `clean` & `site`.

The `clean` lifecycle is designed to clean up all the build directories in preparation for a fresh build.

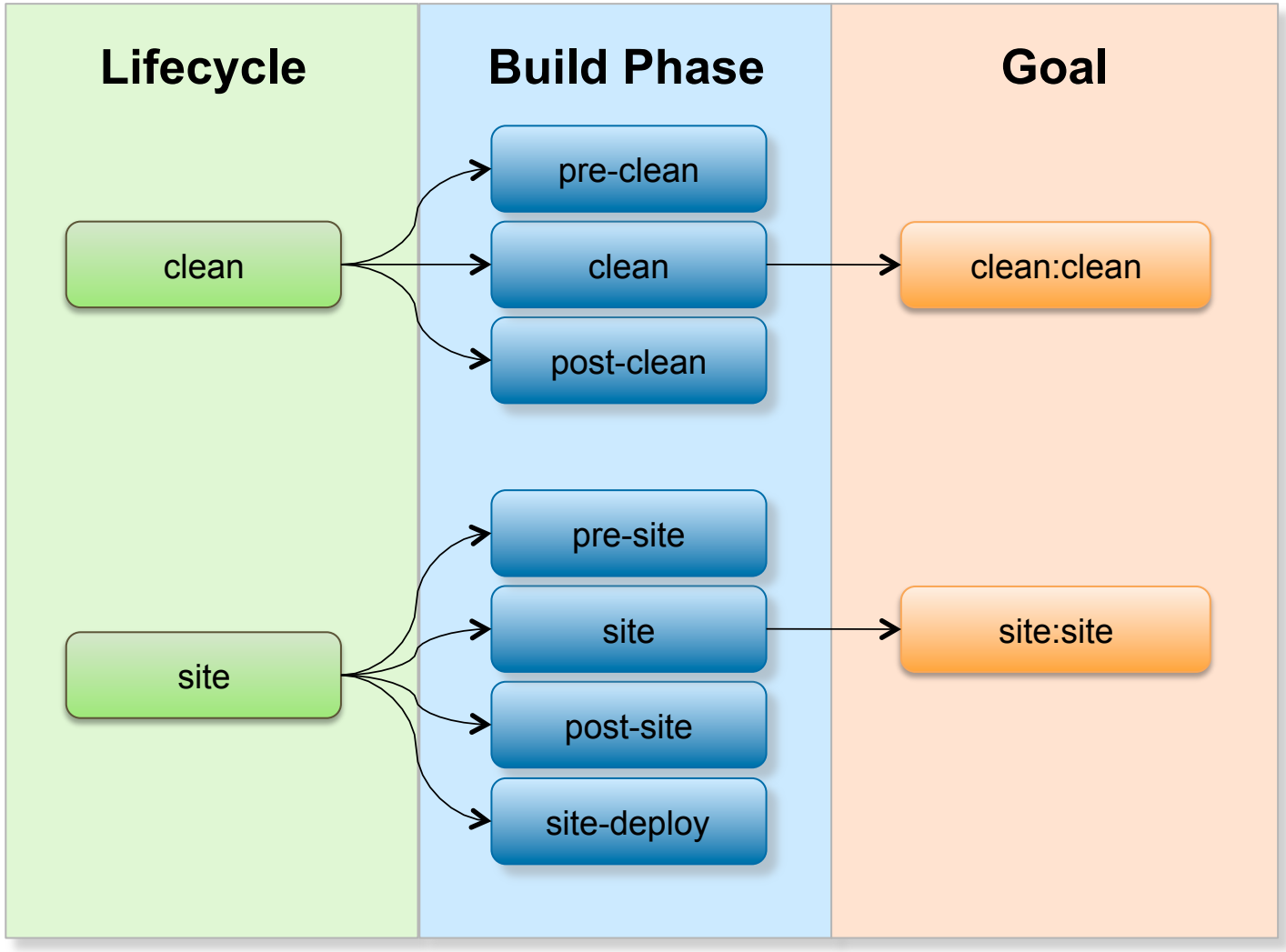


Maven Standard Lifecycles 2/2

The other standard lifecycles are `clean` & `site`.

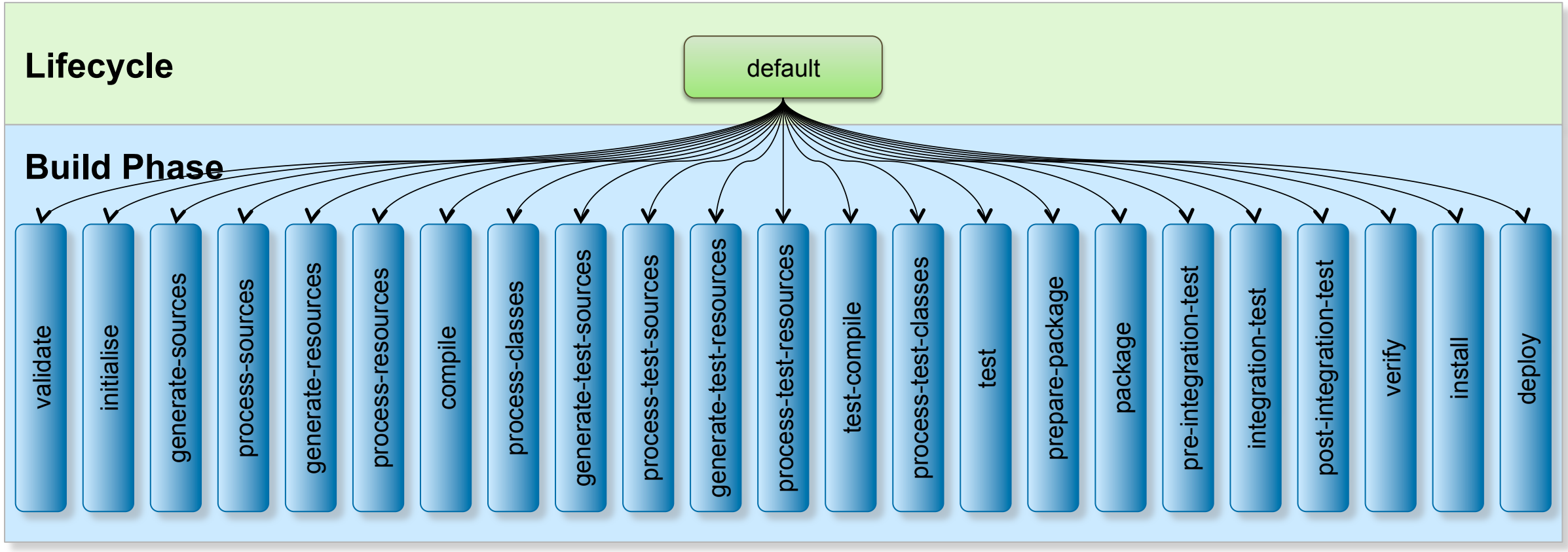
The `clean` lifecycle is designed to clean up all the build directories in preparation for a fresh build.

The `site` lifecycle is designed to perform the steps necessary for building an entire website and deploying it to a server.



Maven Lifecycle: `Default`

The build phases shown previously for the `default` lifecycle are only the most frequently used ones. In fact, this lifecycle contains 23 distinct build phases; however, their use is entirely optional.



Maven

The POM File

Project Object Model (POM) File

The Project Object Model is an XML description of your Java project. This description is stored in one or more `pom.xml` files. At the very minimum, a POM file must provide Maven with the following values:

Project Object Model (POM) File

The Project Object Model is an XML description of your Java project. This description is stored in one or more `pom.xml` files. At the very minimum, a POM file must provide Maven with the following values:

- A `<project>` root element

```
<project>



</project>
```

Project Object Model (POM) File

The Project Object Model is an XML description of your Java project. This description is stored in one or more `pom.xml` files. At the very minimum, a POM file must provide Maven with the following values:

- A `<project>` root element
- A `<modelVersion>` – hardcoded to 4.0.0

```
<project>
  <modelVersion>4.0.0</modelVersion>



</project>
```

Project Object Model (POM) File

The Project Object Model is an XML description of your Java project. This description is stored in one or more `pom.xml` files. At the very minimum, a POM file must provide Maven with the following values:

- A `<project>` root element
- A `<modelVersion>` – hardcoded to 4.0.0
- A `<groupId>` – A unique identifier for the group to which this artifact belongs

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>


</project>
```

Project Object Model (POM) File

The Project Object Model is an XML description of your Java project. This description is stored in one or more `pom.xml` files. At the very minimum, a POM file must provide Maven with the following values:

- A `<project>` root element
- A `<modelVersion>` – hardcoded to 4.0.0
- A `<groupId>` – A unique identifier for the group to which this artifact belongs
- An `<artifactId>` – A unique identifier of this particular project (artifact)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>my-app</artifactId>

</project>
```

Project Object Model (POM) File

The Project Object Model is an XML description of your Java project. This description is stored in one or more `pom.xml` files. At the very minimum, a POM file must provide Maven with the following values:

- A `<project>` root element
- A `<modelVersion>` – hardcoded to 4.0.0
- A `<groupId>` – A unique identifier for the group to which this artifact belongs
- An `<artifactId>` – A unique identifier of this particular project (artifact)
- A `<version>` – The version number for this project's current state

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

Project Object Model (POM) File

The Project Object Model is an XML description of your Java project. This description is stored in one or more `pom.xml` files. At the very minimum, a POM file must provide Maven with the following values:

- A `<project>` root element
- A `<modelVersion>` – hardcoded to 4.0.0
- A `<groupId>` – A unique identifier for the group to which this artifact belongs
- An `<artifactId>` – A unique identifier of this particular project (artifact)
- A `<version>` – The version number for this project's current state

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

These three values taken together form what Maven calls the artifact's ***coordinates***

Maven

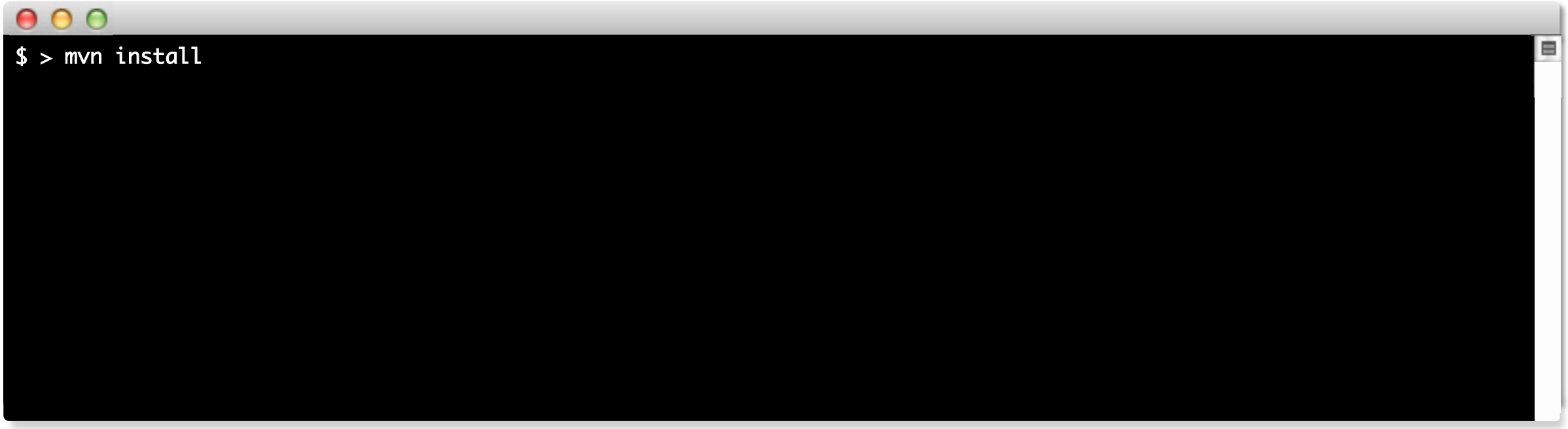
Invoking Maven Functionality

Invoking Build Phase Functionality 1/2

Maven functionality is invoked from the command line using the `mvn` command.

The parameter passed to `mvn` indicates the name of the build phase you wish Maven to run up to.

For instance, if you wish to run everything in the `default` lifecycle ***up to and including*** the `install` build phase, then you would enter:



Invoking Build Phase Functionality 1/2

Maven functionality is invoked from the command line using the `mvn` command.

The parameter passed to `mvn` indicates the name of the build phase you wish Maven to run up to.

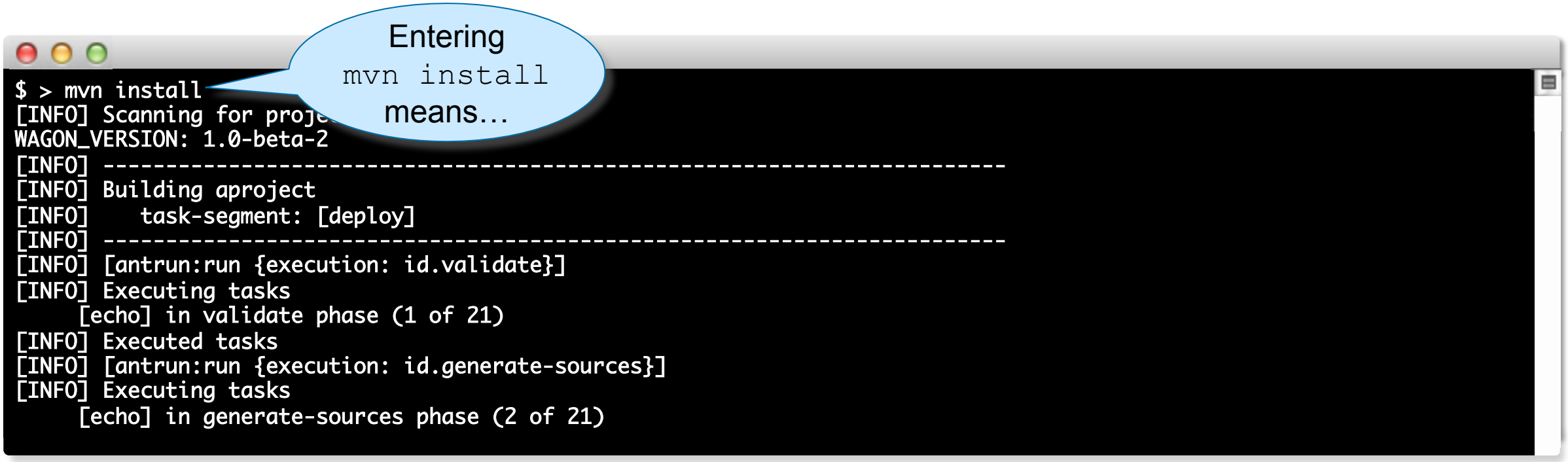
For instance, if you wish to run everything in the `default` lifecycle ***up to and including*** the `install` build phase, then you would enter:

```
$ > mvn install
[INFO] Scanning for projects...
WAGON_VERSION: 1.0-beta-2
[INFO] ------------------------------------------------------------------------
[INFO] Building aproject
[INFO]    task-segment: [deploy]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: id.validate}]
[INFO] Executing tasks
     [echo] in validate phase (1 of 21)
[INFO] Executed tasks
[INFO] [antrun:run {execution: id.generate-sources}]
[INFO] Executing tasks
     [echo] in generate-sources phase (2 of 21)
```

Invoking Build Phase Functionality 2/2

Successful use of Maven is based on having a good understanding of the build phases in the lifecycle you are using, and then knowing how many of those build phases you currently require.

Whenever you invoke any Maven functionality, you will typically be telling Maven to run through a standard sequence of build phases ***up until a certain point***.



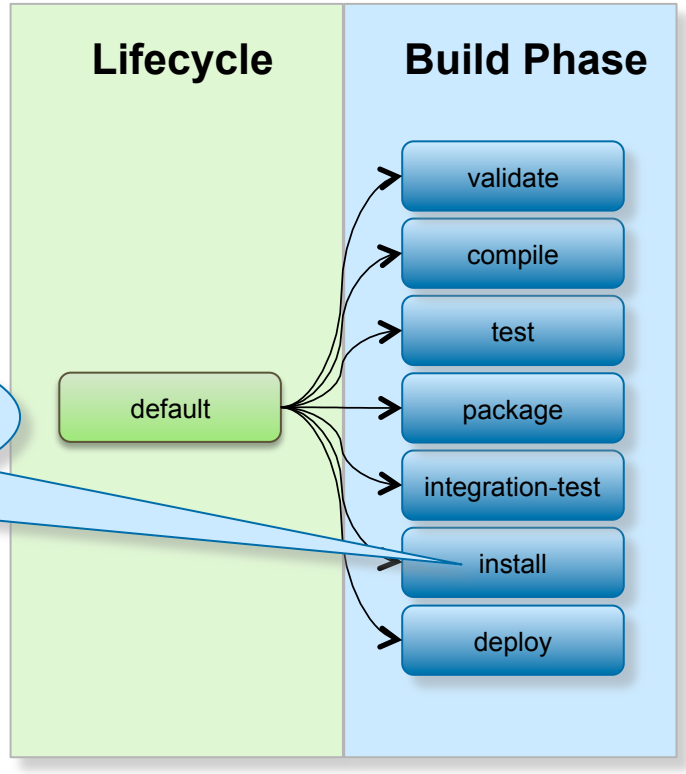
Invoking Build Phase Functionality 2/2

Successful use of Maven is based on having a good understanding of the build phases in the lifecycle you are using, and then knowing how many of those build phases you currently require.

Whenever you invoke any Maven functionality, you will typically be telling Maven to run through a standard sequence of build phases ***up until a certain point***.

```
$ > mvn install
[INFO] Scanning for projects...
WAGON_VERSION: 1.0-beta-2
[INFO] ------------------------------------------------------------
[INFO] Building aproject
[INFO]    task-segment: [de
[INFO] ------------------
[INFO] [antrun:run {executio
[INFO] Executing tasks
     [echo] in validate phase (1 of 21)
[INFO] Executed tasks
[INFO] [antrun:run {execution: id.generate-sources}]
[INFO] Executing tasks
     [echo] in generate-sources phase (2 of 21)
```

Run all the build phases in the `default` lifecycle ***up to and including*** the `install` phase

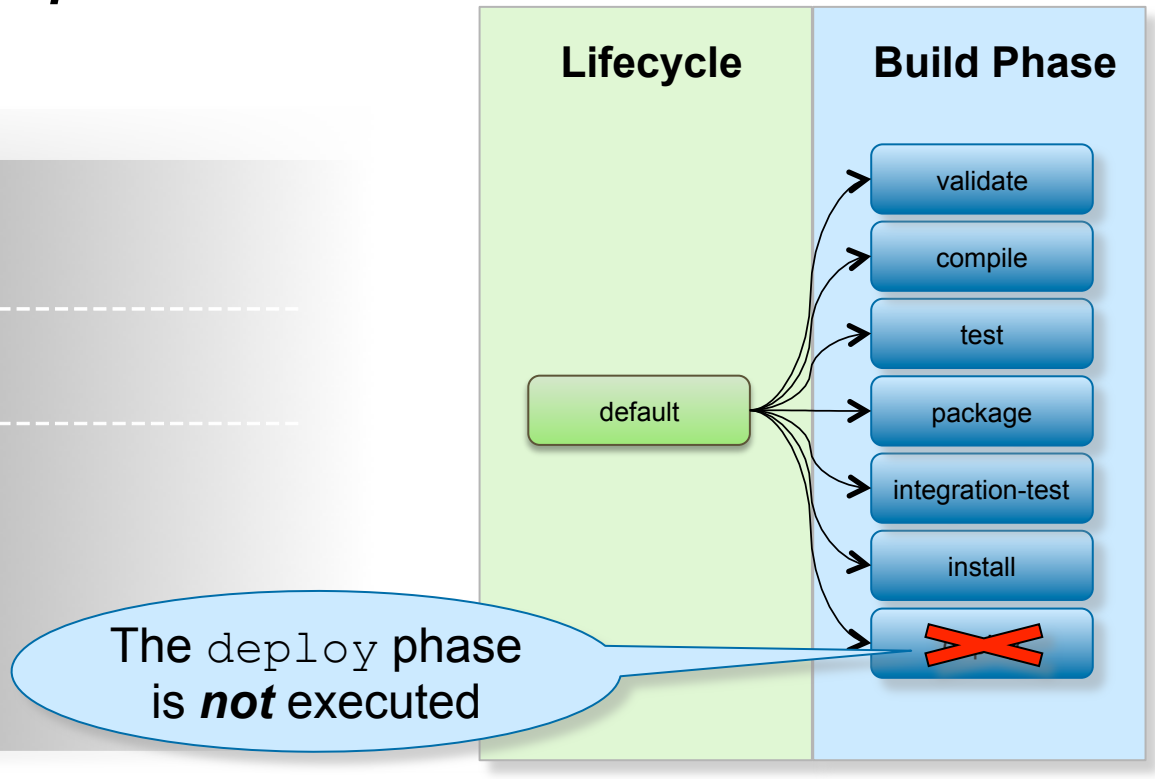


Invoking Build Phase Functionality 2/2

Successful use of Maven is based on having a good understanding of the build phases in the lifecycle you are using, and then knowing how many of those build phases you currently require.

Whenever you invoke any Maven functionality, you will typically be telling Maven to run through a standard sequence of build phases ***up until a certain point***.

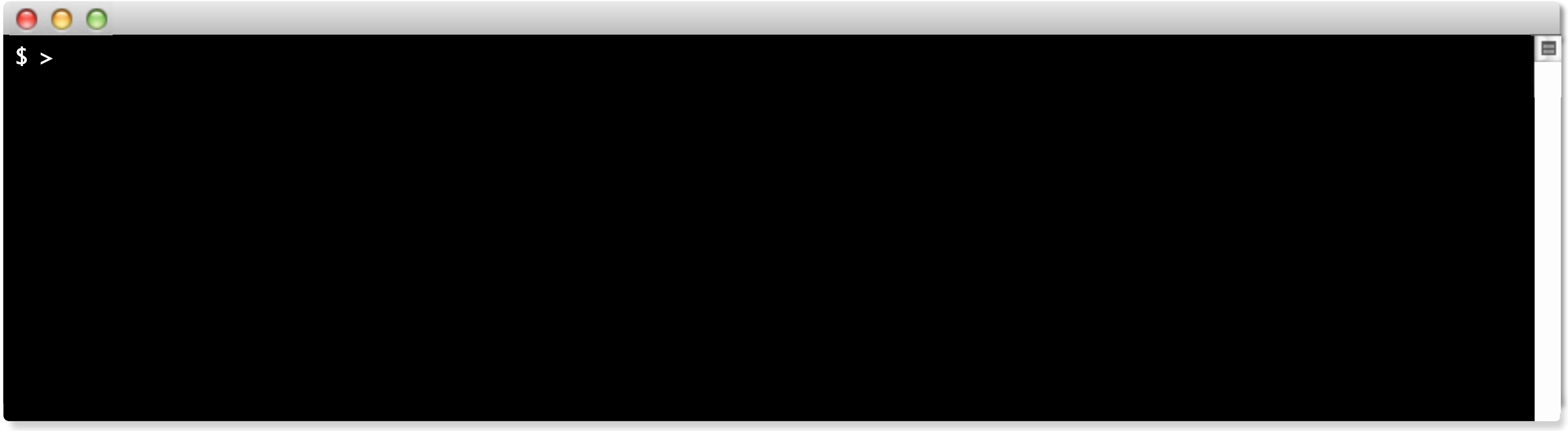
```
$ > mvn install
[INFO] Scanning for projects...
WAGON_VERSION: 1.0-beta-2
[INFO] ------------------------------------------------------------------------
[INFO] Building aproject
[INFO]    task-segment: [deploy]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: id.validate}]
[INFO] Executing tasks
     [echo] in validate phase (1 of 21)
[INFO] Executed tasks
[INFO] [antrun:run {execution: id.generate-sources}]
[INFO] Executing tasks
     [echo] in generate-sources phase (2 of 21)
```



Invoking Functionality from Multiple Lifecycles 1/2

Maven can also run multiple sequences of build phases belonging to different lifecycles.

For instance is it quite common to issue the following command: `mvn clean install`

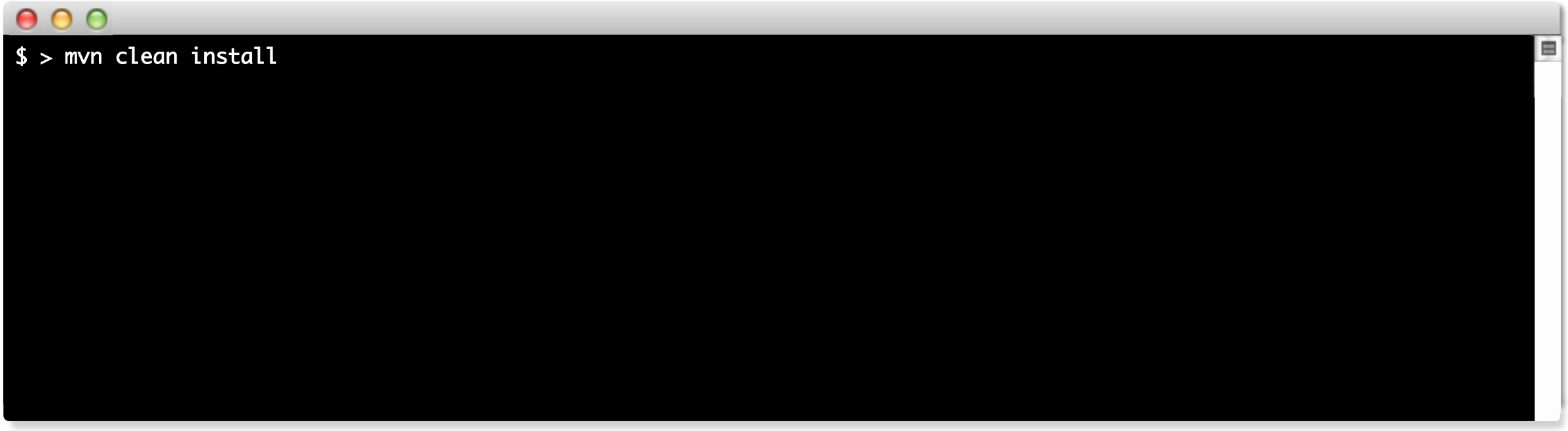


Invoking Functionality from Multiple Lifecycles 1/2

Maven can also run multiple sequences of build phases belonging to different lifecycles.

For instance is it quite common to issue the following command: `mvn clean install`

Entering this command means two things

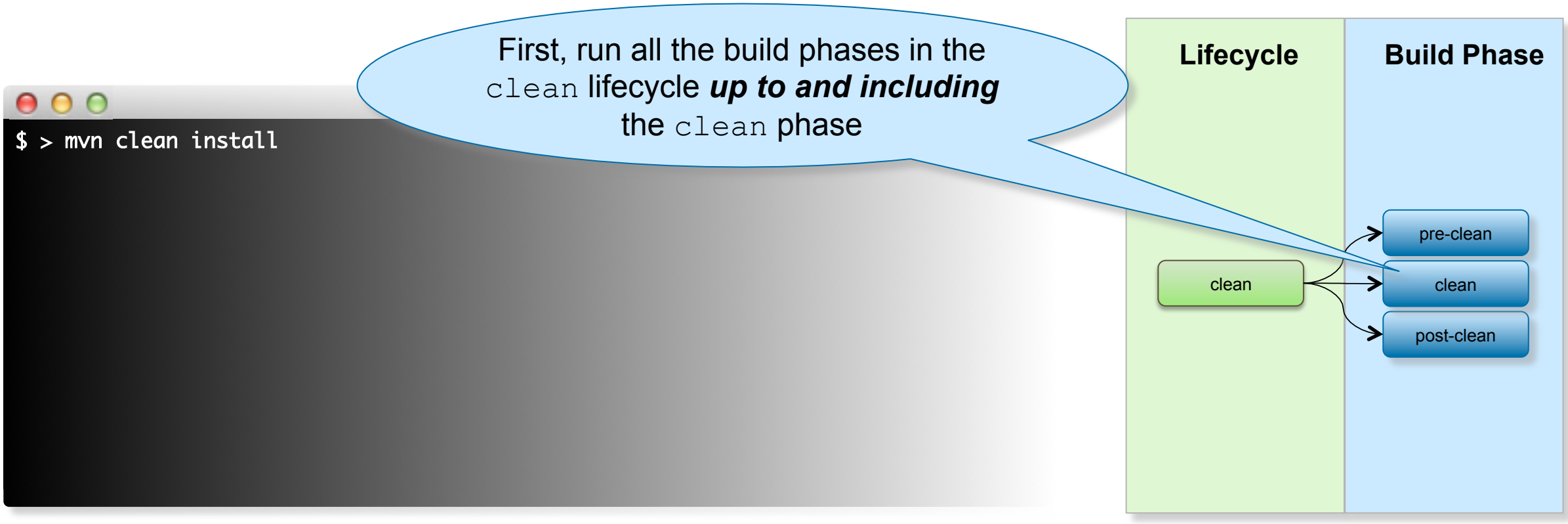


Invoking Functionality from Multiple Lifecycles 1/2

Maven can also run multiple sequences of build phases belonging to different lifecycles.

For instance is it quite common to issue the following command: `mvn clean install`

Entering this command means two things



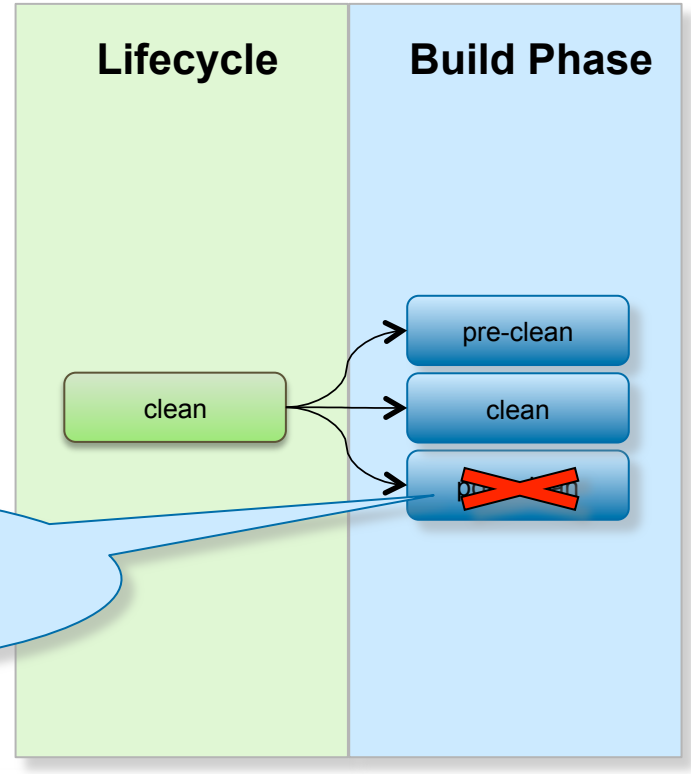
Invoking Functionality from Multiple Lifecycles 1/2

Maven can also run multiple sequences of build phases belonging to different lifecycles.

For instance is it quite common to issue the following command: `mvn clean install`

Entering this command means two things

```
$ > mvn clean install
```

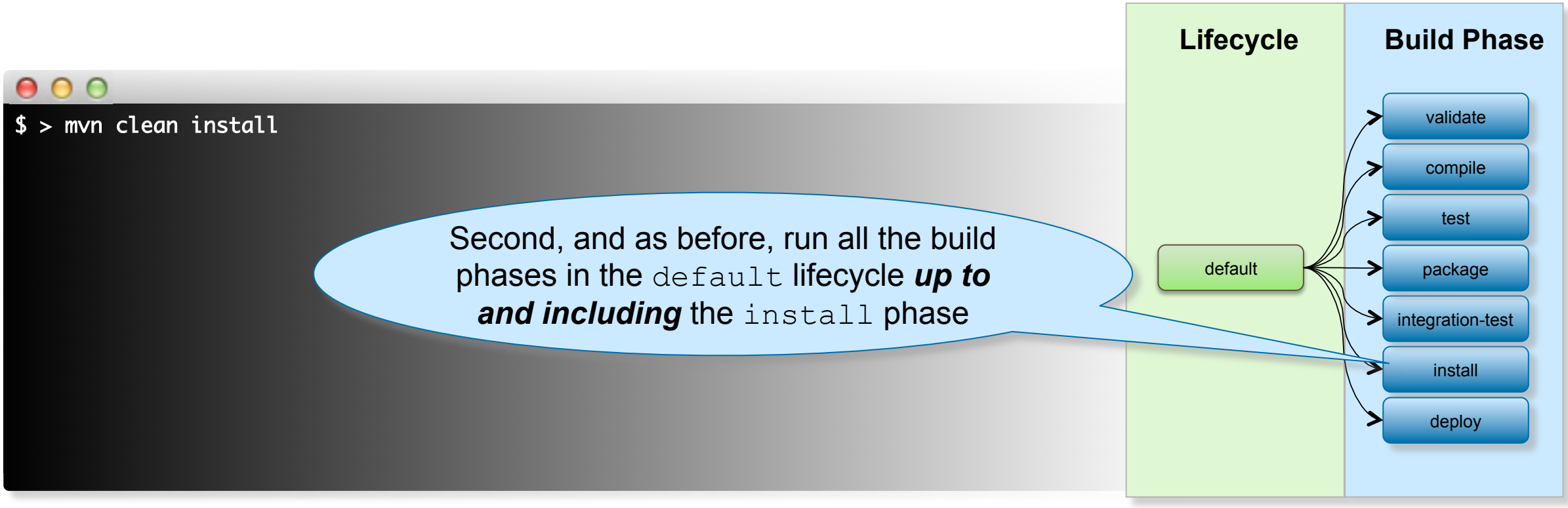


Invoking Functionality from Multiple Lifecycles 2/2

Maven can also run multiple sequences of build phases belonging to different lifecycles.

For instance is it quite common to issue the following command: `mvn clean install`

Entering this command means two things

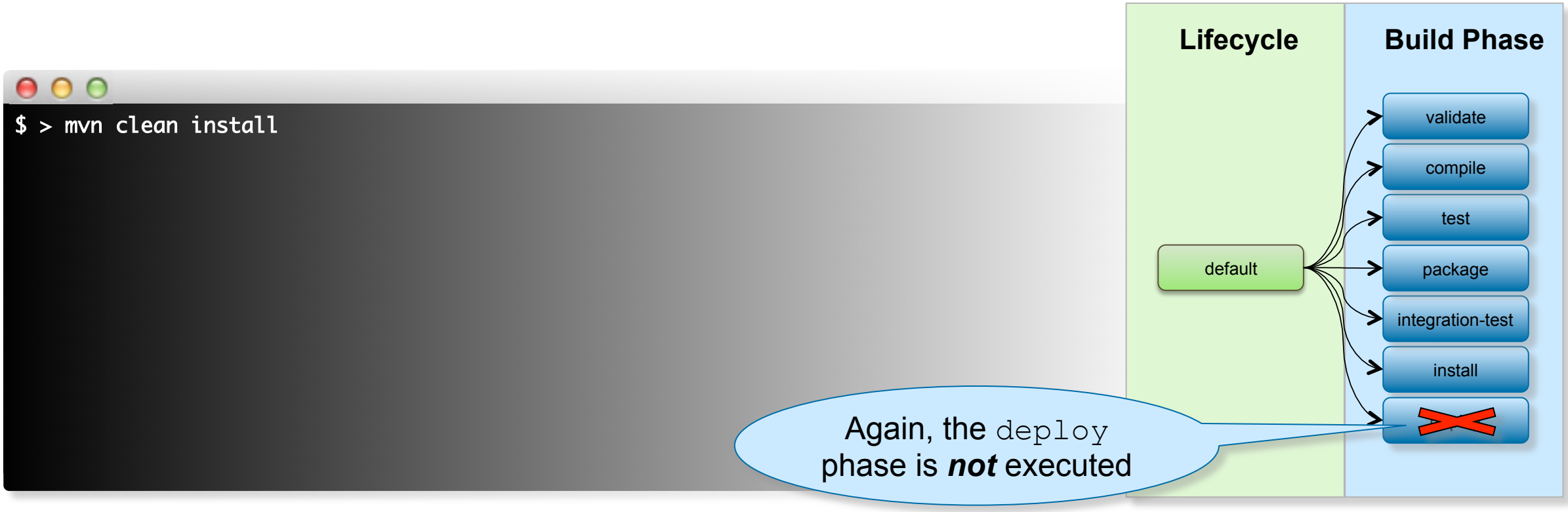


Invoking Functionality from Multiple Lifecycles 2/2

Maven can also run multiple sequences of build phases belonging to different lifecycles.

For instance is it quite common to issue the following command: `mvn clean install`

Entering this command means two things

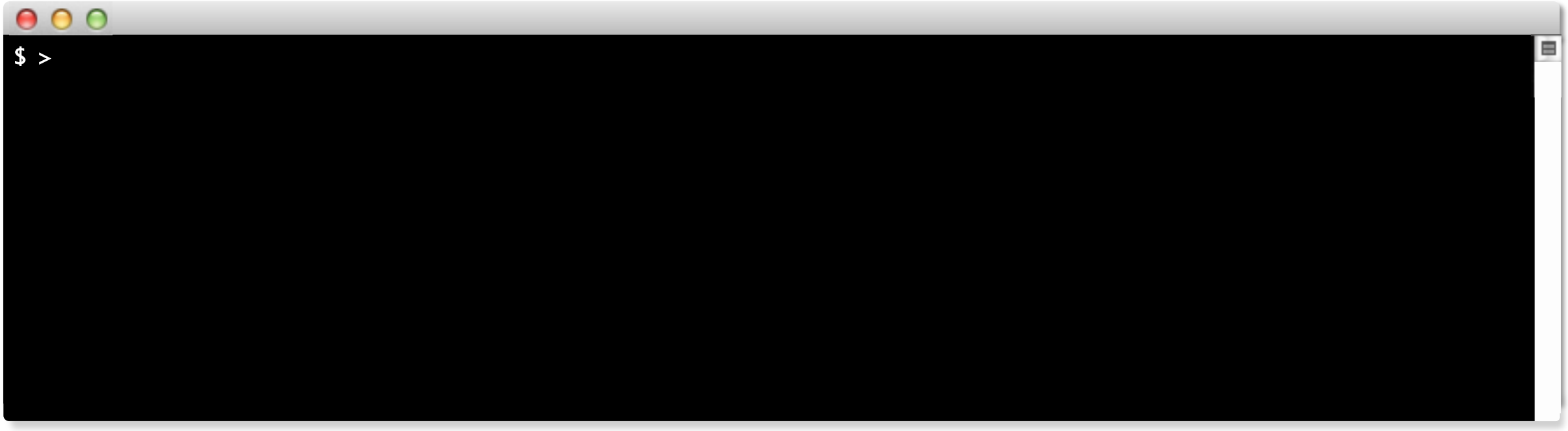


Maven

Creating a Basic Maven Project

Creating a Maven Project: Invocation

In addition to specifying build phase names as parameters to the `mvn` command, you can directly specify a plugin-id and a goal-id. This allows you to run the functionality of a specific goal within a plug-in without that goal needing to be bound to a build phase.



Creating a Maven Project: Invocation

In addition to specifying build phase names as parameters to the `mvn` command, you can directly specify a plugin-id and a goal-id. This allows you to run the functionality of a specific goal within a plug-in without that goal needing to be bound to a build phase.

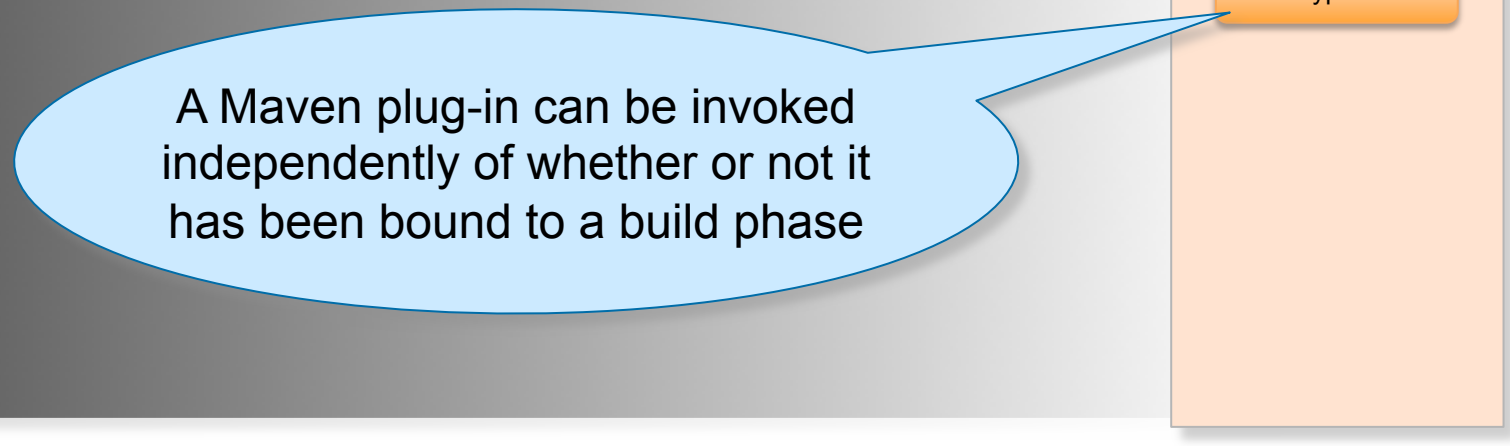
In the case of creating a Maven project, we can invoke the `create` goal belonging to the `archetype` plug-in. This is an example of a goal that is ***never*** bound to a build phase:

Creating a Maven Project: Invocation

In addition to specifying build phase names as parameters to the `mvn` command, you can directly specify a plugin-id and a goal-id. This allows you to run the functionality of a specific goal within a plug-in without that goal needing to be bound to a build phase.

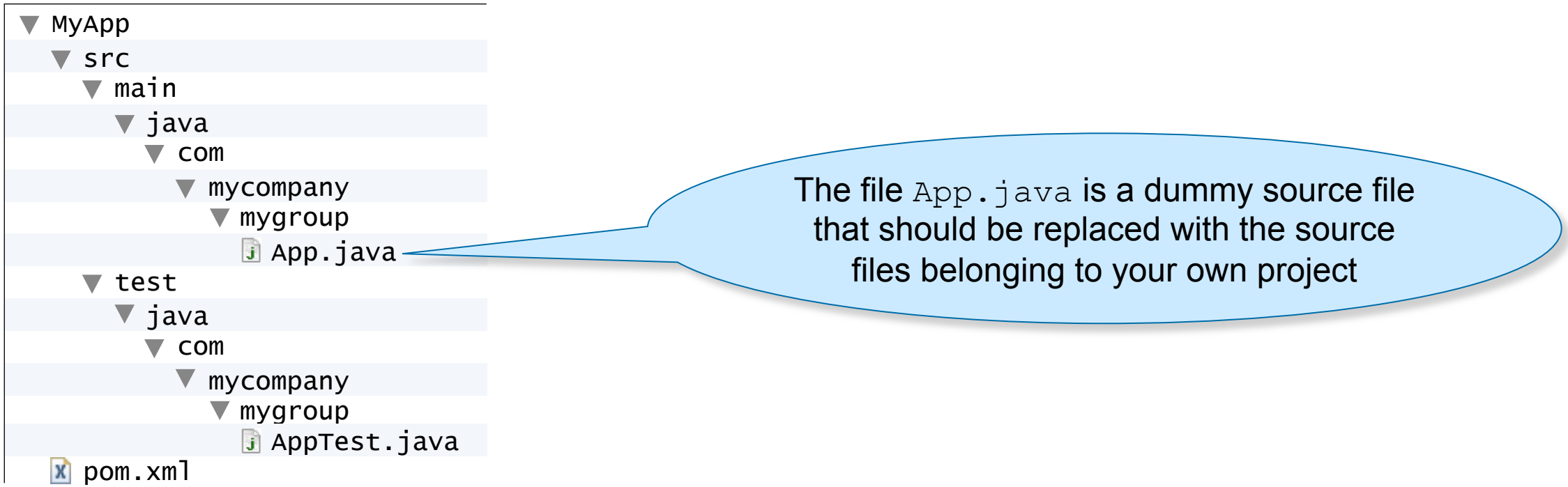
In the case of creating a Maven project, we can invoke the `create` goal belonging to the `archetype` plug-in. This is an example of a goal that is ***never*** bound to a build phase:

```
$ > mvn archetype:create -DarchetypeGroupId=org.apache.maven.archetypes \
      -DgroupId=com.mycompany.mygroup -DartifactId=MyApp
```



Creating a Maven Project: Directory Structure

As a result of directly running the `archetype:create` goal (using the parameters seen on the previous page), the following directory structure will be created under your Maven home directory:



Creating a Maven Project: Generated POM File

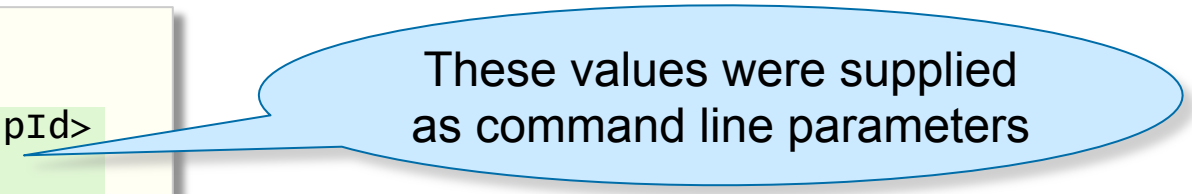
The `archetype:create` plug-in also creates a basic `pom.xml` file for you. This contains both the parameters you supplied at the command line and some sensible defaults.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>MyApp</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>
```

Creating a Maven Project: Generated POM File

The `archetype:create` plug-in also creates a basic `pom.xml` file for you. This contains both the parameters you supplied at the command line and some sensible defaults.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>MyApp</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>
```



Creating a Maven Project: Generated POM File

The `archetype:create` plug-in also creates a basic `pom.xml` file for you. This contains both the parameters you supplied at the command line and some sensible defaults.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>MyApp</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>
```

Sensible default values assume that the project:

- Will be packaged as a `jar` file
- Is currently at version 1
- Depends on `junit` for testing

Creating a Maven Project: Generated POM File

The `archetype:create` plug-in also creates a basic `pom.xml` file for you. This contains both the parameters you supplied at the command line and some sensible defaults.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>MyApp</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>
```

The dependency on `junit` exists only during the build phase `test`

Maven

Multi-module Projects

Multi-module Projects 1/5

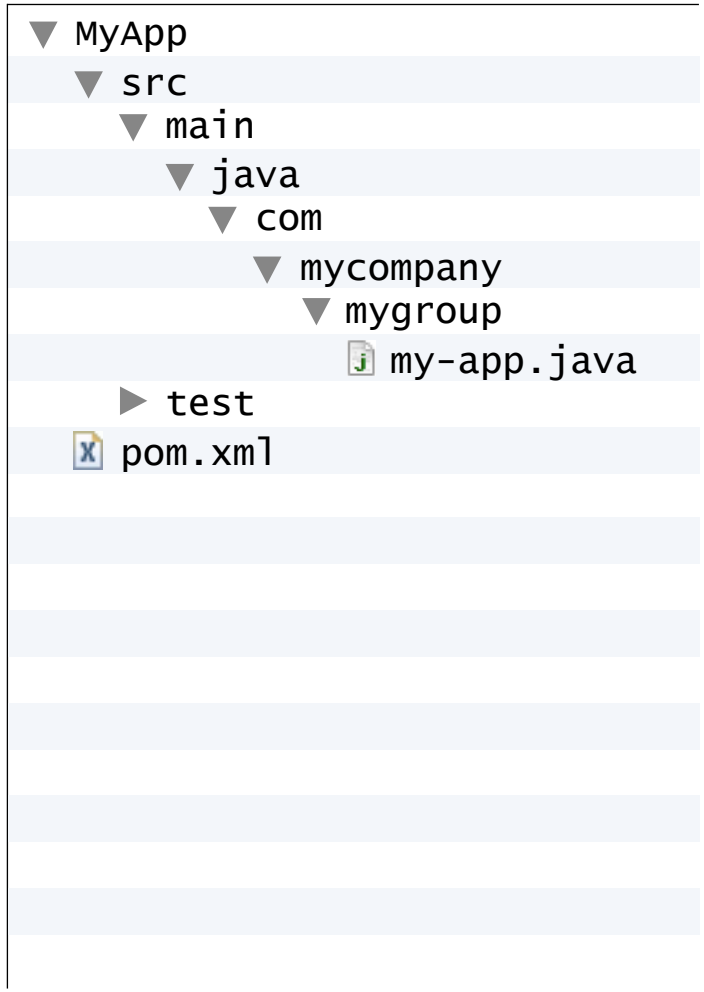
The project dependency shown here is one in which the artifact upon which this project depends is identified by its coordinates (`groupId` + `artifactId` + `version` = `coordinates`)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>MyApp</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>
```

If this artifact does not exist in the local repository, Maven's Dependency Management Model will download it from one of the remote repositories listed in `settings.xml`

Multi-module Projects 2/5

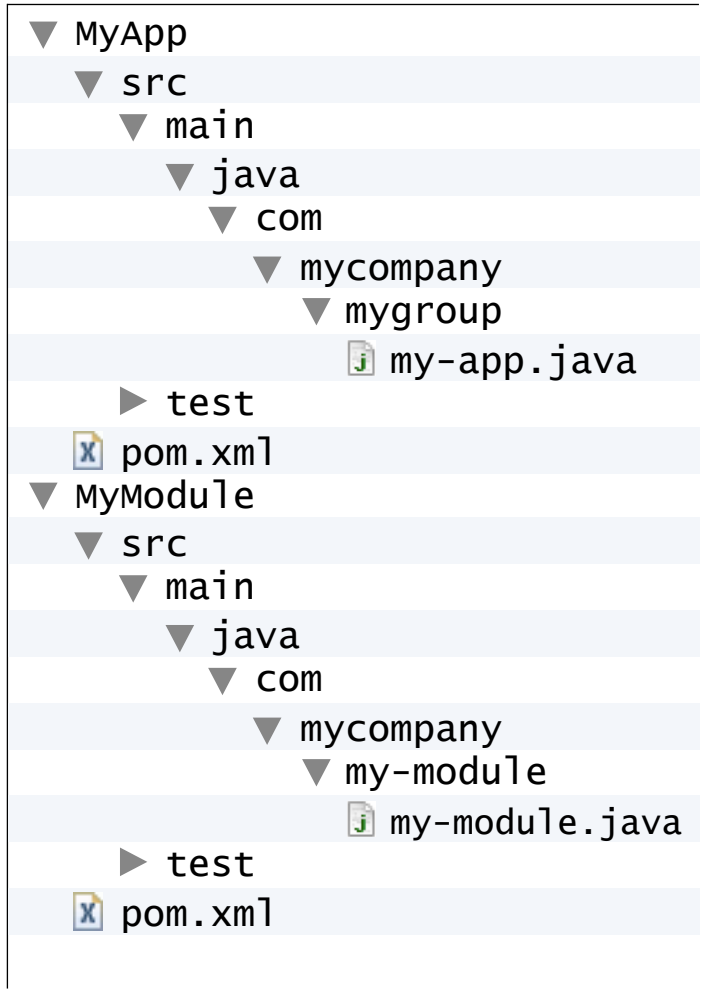
In real life, projects contain multiple modules.



Multi-module Projects 2/5

In real life, projects contain multiple modules.

Here we have added a second module called `myModule`.



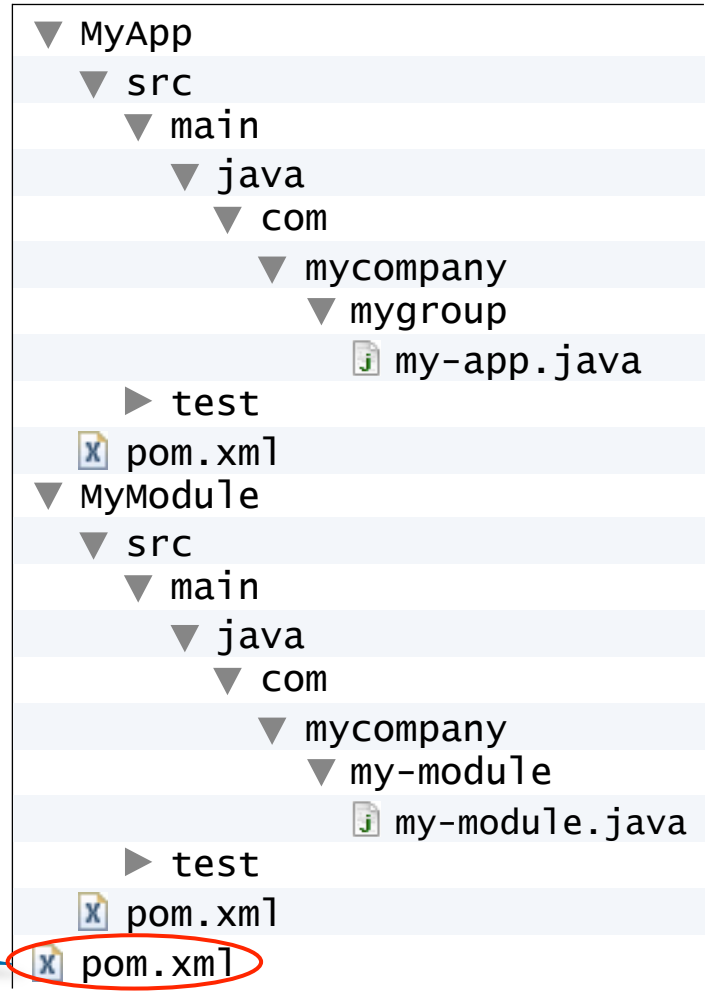
Multi-module Projects 2/5

In real life, projects contain multiple modules.

Here we have added a second module called `myModule`.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>MavenExample</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Example</name>
  <url>http://maven.apache.org</url>

  <modules>
    <module>MyApp</module>
    <module>MyModule</module>
  </modules>
</project>
```

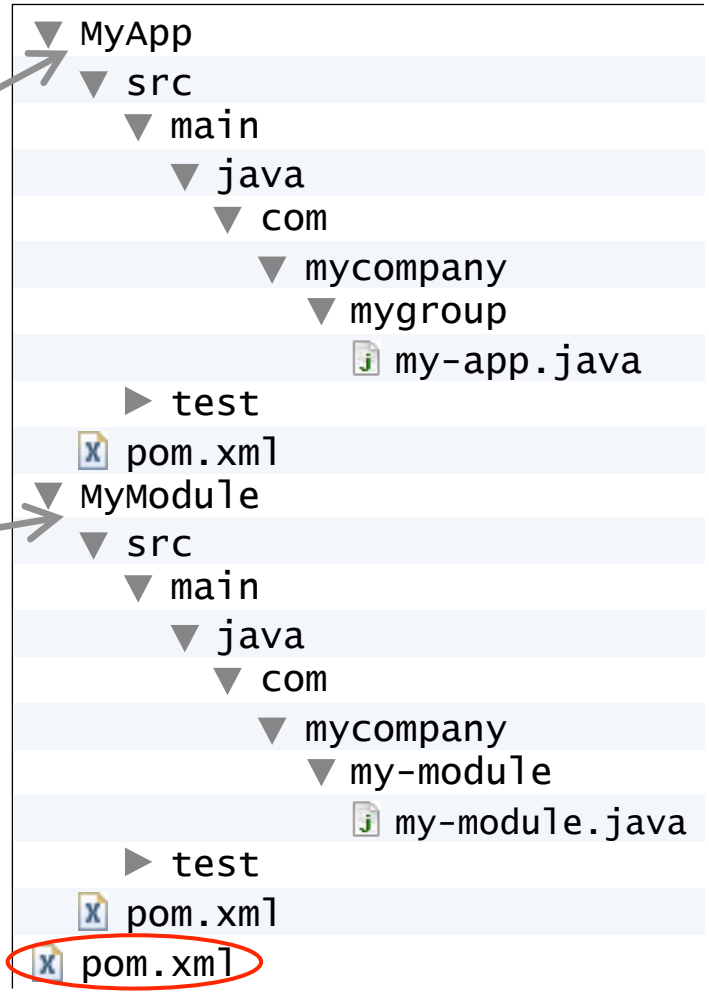
The first step is to create a top-level POM file in the directory above the two modules



Multi-module Projects 3/5

The top-level POM file now declares that this project is composed of two modules.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>MavenExample</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Example</name>
  <url>http://maven.apache.org</url>

  <modules>
    <module>MyApp</module>
    <module>MyModule</module>
  </modules>
</project>
```

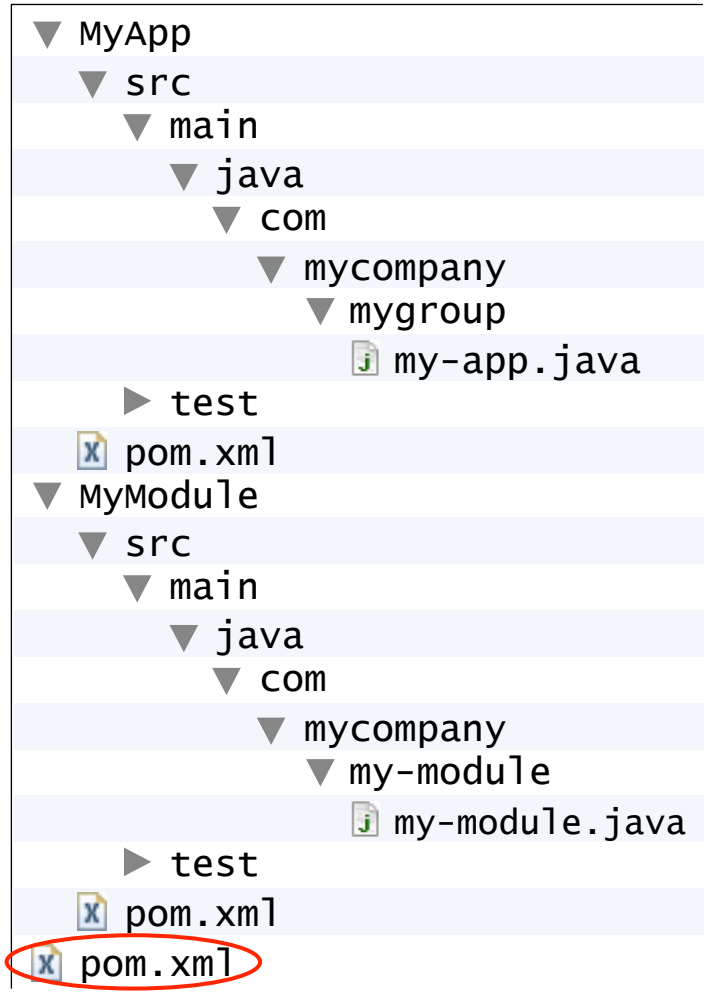


Multi-module Projects 3/5

The top-level POM file now declares that this project is composed of two modules.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.mygroup</groupId>
  <artifactId>MavenExample</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Example</name>
  <url>http://maven.apache.org</url>

  <modules>
    <module>MyApp</module>
    <module>MyModule</module>
  </modules>
</project>
```

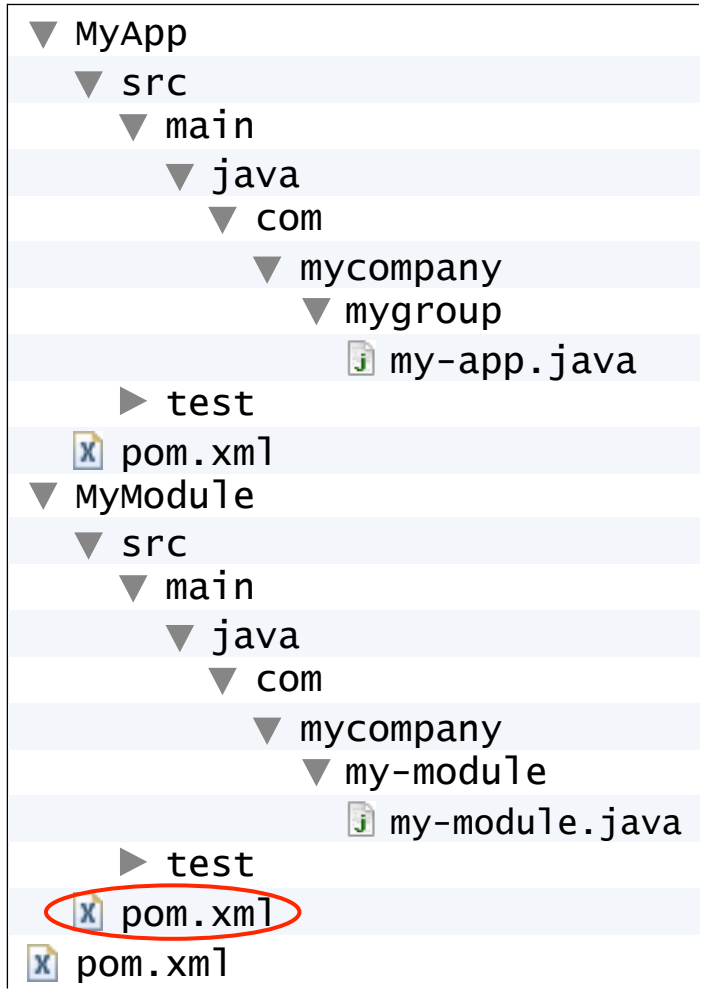
The packaging type must be set to `pom` to indicate a multi-module project



Multi-module Projects 4/5

The POM files belonging to `MyApp` and `MyModule` now need to declare that their parent is the top-level POM file.

```
<project>
  <parent>
     <groupId>com.mycompany.mygroup</groupId>
     <artifactId>MavenExample</artifactId>
     <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>MyModule</artifactId>
  <packaging>jar</packaging>
</project>
```

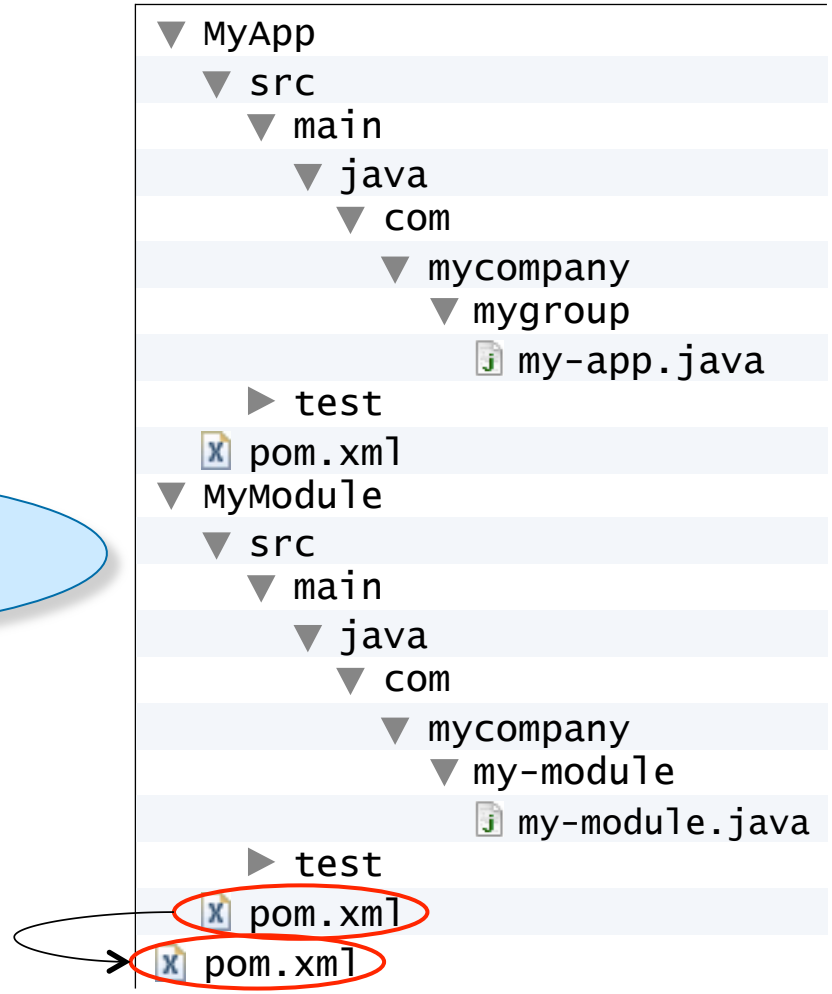


Multi-module Projects 4/5

The POM files belonging to `MyApp` and `MyModule` now need to declare that their parent is the top-level POM file.

```
<project>
  <parent>
     <groupId>com.mycompany.mygroup</groupId>
     <artifactId>MavenExample</artifactId>
     <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>MyModule</artifactId>
  <packaging>jar</packaging>
</project>
```

The parent artifact is identified by its coordinates

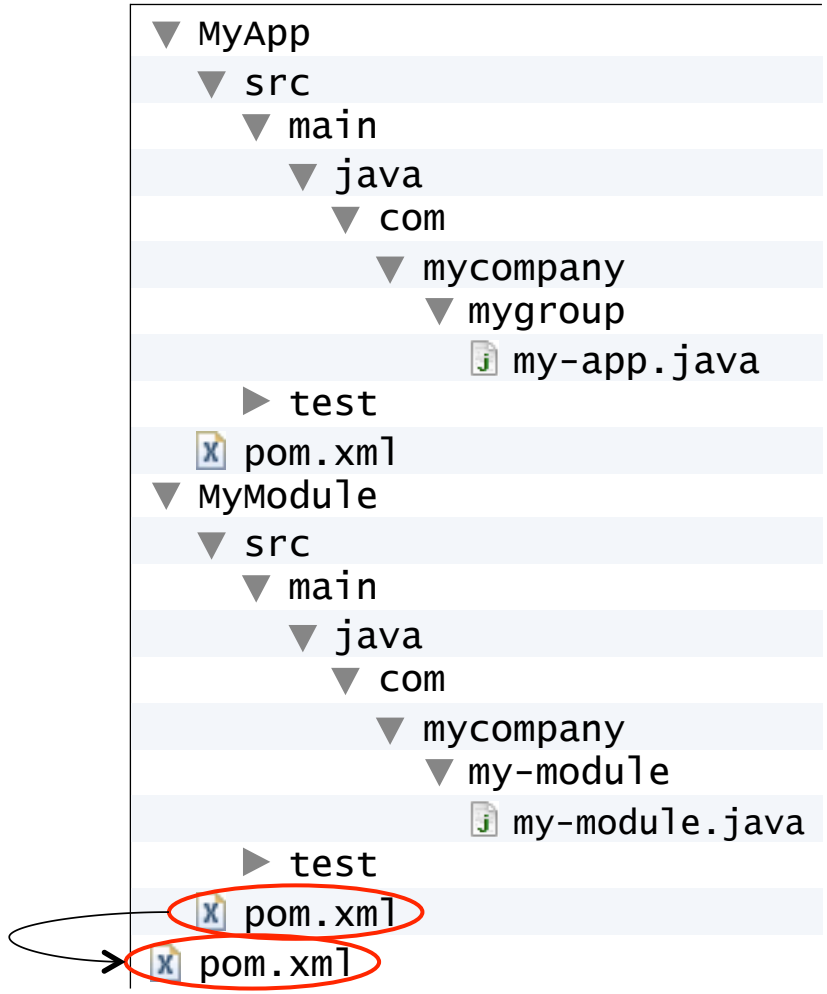


Multi-module Projects 4/5

The POM files belonging to `MyApp` and `MyModule` now need to declare that their parent is the top-level POM file.

```
<project>
  <parent>
     <groupId>com.mycompany.mygroup</groupId>
     <artifactId>MavenExample</artifactId>
     <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>MyModule</artifactId>
  <packaging>jar</packaging>
</project>
```

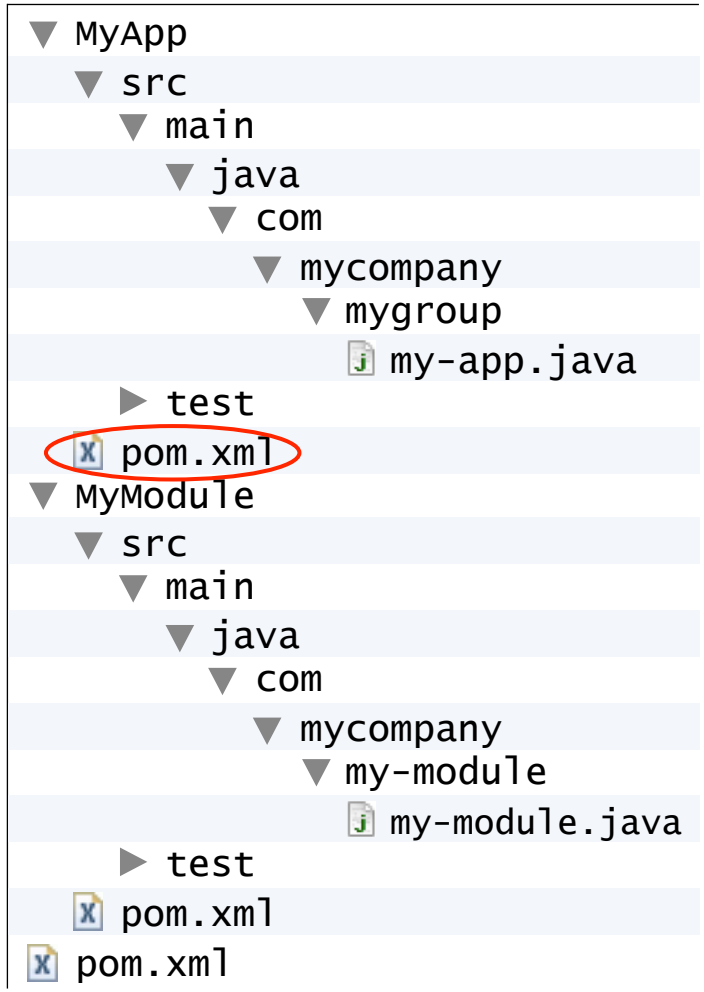
Due to inheritance, only values that are different from the parent POM need to be specified



Multi-module Projects 5/5

Since we have written `MyApp` to be dependent upon `MyModule`, this relationship must also be specified in `MyApp`'s POM file.

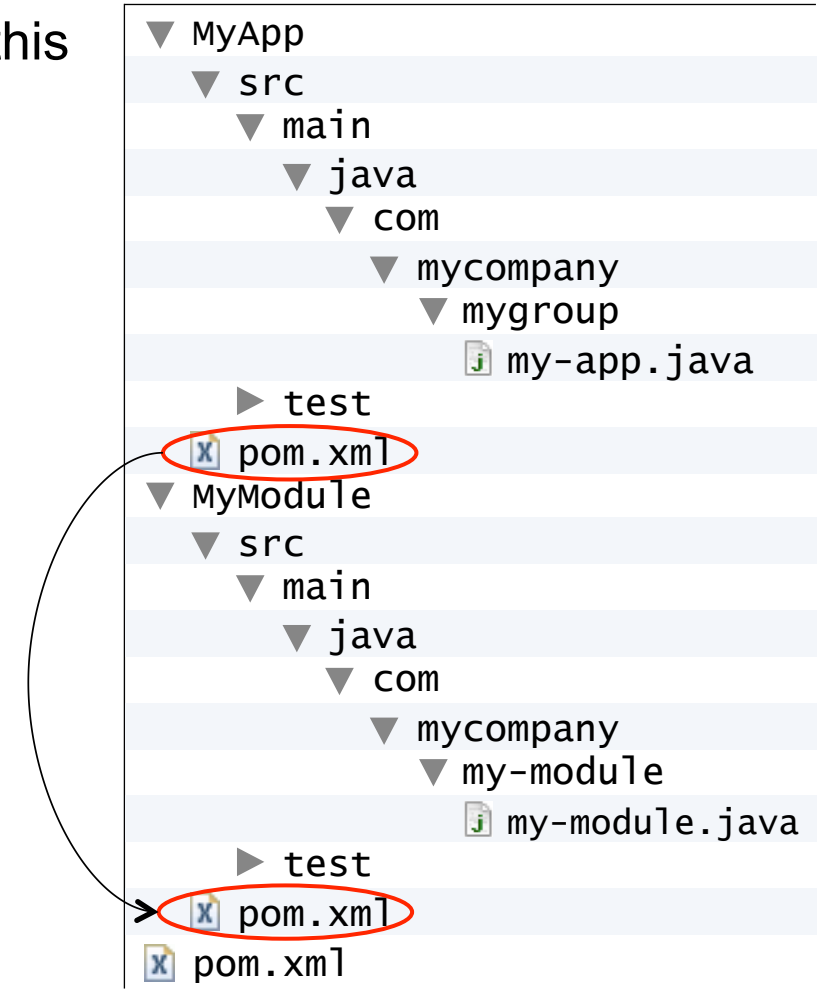
```
<project>
  <parent>
     <groupId>com.mycompany.mygroup</groupId>
     <artifactId>MavenExample</artifactId>
     <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>MyApp</artifactId>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>com.mycompany.mygroup</groupId>
      <artifactId>MyModule</artifactId>
      <version>1</version>
    </dependency>
  </dependencies>
</project>
```



Multi-module Projects 5/5

Since we have written `MyApp` to be dependent upon `MyModule`, this relationship must also be specified in `MyApp`'s POM file.

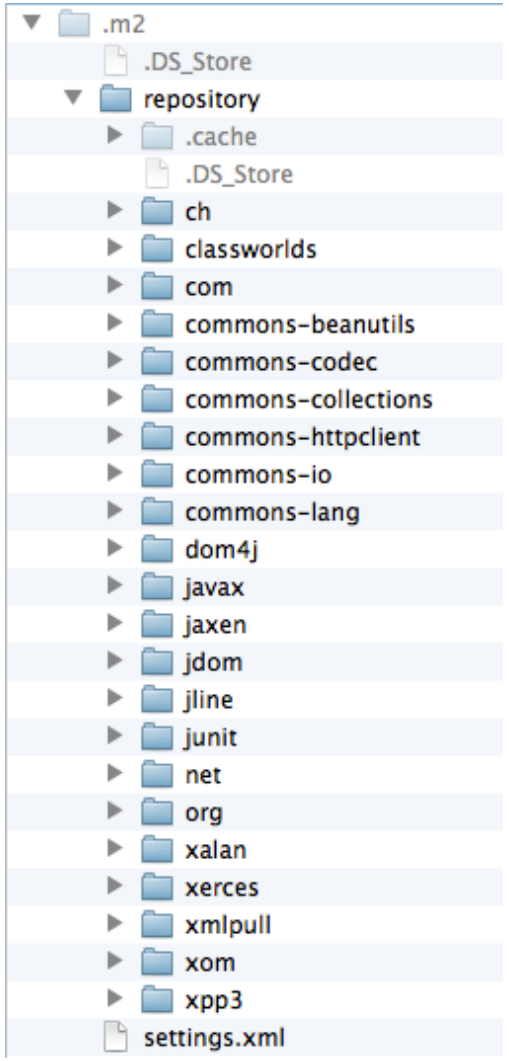
```
<project>
  <parent>
     <groupId>com.mycompany.mygroup</groupId>
     <artifactId>MavenExample</artifactId>
     <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>MyApp</artifactId>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>com.mycompany.mygroup</groupId>
      <artifactId>MyModule</artifactId>
      <version>1</version>
    </dependency>
  </dependencies>
</project>
```



Maven

Local Repository File System Structure

Local Maven Repository



Under your home directory, you will see a directory called `.m2`

Depending on the operating system you are using and the file system settings, this directory may be hidden from normal view.

Under this is the `repository` directory which holds both the software copied from remote repositories, and the `settings.xml` file that holds the list of all repositories known to Maven.

Summary

Maven: Summary 1/2

- Maven a framework for handling the tasks involved in managing and building a Java Project.
- Maven assumes that all Java projects can be built using a standardised sequence of build tasks
- Such a standardised sequence of tasks is known as a ***Lifecycle***
- Each lifecycle is composed of one or more ***Build Phases***
- A build phase can have zero or more ***Goals*** bound to it
- A build phase to which zero goals are bound is automatically skipped
- Goal functionality is implemented through Maven's standardised plug-in API
- A Maven plug-in is known as a ***Mojo***
- Build tasks are invoked from the command line using the `mvn` command followed by:
  - The name of a build phase. Maven will then run all build phases in that lifecycle ***up to and including*** the named build phase.
  - A goal name given in the form `<plugin-id>:<goal-id> <optional_parameters>`
  - Some combination of build phase names and/or goal names

Maven: Summary 2/2

- Maven projects are described using a ***Project Object Model*** (or POM)
- The POM definition is split into two parts
  - Part is hard-coded within Maven's functionality
  - Part is user defined in one or more XML files called `pom.xml`
- Successful use of Maven is centred on the developer's understanding of
  - The standardised POM used by Maven
  - How this standardised POM is implemented as a lifecycle (E.G. the `default` lifecycle)
  - How to construct a `pom.xml` file such that your Java Project fits into Maven's standard POM
- Maven supports the definition of very large and complex project object models by specifying a hierarchy of `pom.xml` files.
- In the absence of user specified values, Maven will use sensible defaults
- Maven's Dependency Management Model resolves artifact dependencies either from the local or remote repositories
- Artifacts copied from remote repositories are stored in a local repository `~/.m2/repository/`

© 2013 SAP AG. All rights reserved

© 2013 SAP AG. Alle Rechte vorbehalten.