



# OOP with Java

## 30. Building with Maven

Thomas Weise · 汤卫思

tweise@hfu.edu.cn · <http://iao.hfu.edu.cn>

Hefei University, South Campus 2  
Faculty of Computer Science and Technology  
Institute of Applied Optimization  
230601 Shushan District, Hefei, Anhui, China  
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区  
计算机科学与技术系  
应用优化研究所  
中国 安徽省 合肥市 蜀山区 230601  
经济技术开发区 锦绣大道99号

- 1 Introduction
- 2 Maven Basics
- 3 Dependencies, Javadoc, Executable, More Infos
- 4 Maven Build with JUnit Tests
- 5 Summary



website

- We have learned a lot of ways to structure our code

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage
- We can put the API specifying interfaces in one library and the implementation of the API into another library, for instance

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage
- We can put the API specifying interfaces in one library and the implementation of the API into another library, for instance
- Projects will naturally end up using a lot of libraries

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage
- We can put the API specifying interfaces in one library and the implementation of the API into another library, for instance
- Projects will naturally end up using a lot of libraries
- As software grows, develops, and is maintained, there will be many versions of these libraries, new versions introducing new features and fixing bugs



- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage
- We can put the API specifying interfaces in one library and the implementation of the API into another library, for instance
- Projects will naturally end up using a lot of libraries
- As software grows, develops, and is maintained, there will be many versions of these libraries, new versions introducing new features and fixing bugs
- A library may, in turn, depend on (specific versions) of other libraries

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage
- We can put the API specifying interfaces in one library and the implementation of the API into another library, for instance
- Projects will naturally end up using a lot of libraries
- As software grows, develops, and is maintained, there will be many versions of these libraries, new versions introducing new features and fixing bugs
- A library may, in turn, depend on (specific versions) of other libraries, which then depend on yet other libraries

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage
- We can put the API specifying interfaces in one library and the implementation of the API into another library, for instance
- Projects will naturally end up using a lot of libraries
- As software grows, develops, and is maintained, there will be many versions of these libraries, new versions introducing new features and fixing bugs
- A library may, in turn, depend on (specific versions) of other libraries, which then depend on yet other libraries
- How do we manage all of that?

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage
- We can put the API specifying interfaces in one library and the implementation of the API into another library, for instance
- Projects will naturally end up using a lot of libraries
- As software grows, develops, and is maintained, there will be many versions of these libraries, new versions introducing new features and fixing bugs
- A library may, in turn, depend on (specific versions) of other libraries, which then depend on yet other libraries
- How do we manage all of that? How can we achieve that our team members all work with the same versions of the required libraries?

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage
- We can put the API specifying interfaces in one library and the implementation of the API into another library, for instance
- Projects will naturally end up using a lot of libraries
- As software grows, develops, and is maintained, there will be many versions of these libraries, new versions introducing new features and fixing bugs
- A library may, in turn, depend on (specific versions) of other libraries, which then depend on yet other libraries
- How do we manage all of that? How can we achieve that our team members all work with the same versions of the required libraries?
- We need help.

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage
- We can put the API specifying interfaces in one library and the implementation of the API into another library, for instance
- Projects will naturally end up using a lot of libraries
- As software grows, develops, and is maintained, there will be many versions of these libraries, new versions introducing new features and fixing bugs
- A library may, in turn, depend on (specific versions) of other libraries, which then depend on yet other libraries
- How do we manage all of that? How can we achieve that our team members all work with the same versions of the required libraries?
- We need help. Help by a tool.

- We have learned a lot of ways to structure our code
- We can divide it into methods, classes, and packages
- We can create interfaces to specify an API and then cleanly separate the API implementation from the API usage
- We can put the API specifying interfaces in one library and the implementation of the API into another library, for instance
- Projects will naturally end up using a lot of libraries
- As software grows, develops, and is maintained, there will be many versions of these libraries, new versions introducing new features and fixing bugs
- A library may, in turn, depend on (specific versions) of other libraries, which then depend on yet other libraries
- How do we manage all of that? How can we achieve that our team members all work with the same versions of the required libraries?
- We need help. Help by a tool. Maven is the tool.

- Maven is maybe the most widely-used project build and dependency management tool in Java



- Maven is maybe the most widely-used project build and dependency management tool in Java
- It allows you to specify which other software your project depends on

- Maven is maybe the most widely-used project build and dependency management tool in Java
- It allows you to specify which other software your project depends on, which is then automatically downloaded and installed during the build process

- Maven is maybe the most widely-used project build and dependency management tool in Java
- It allows you to specify which other software your project depends on, which is then automatically downloaded and installed during the build process
- Maven can build your project and generate archives, documentation, a project website, and other artifacts

- Maven is maybe the most widely-used project build and dependency management tool in Java
- It allows you to specify which other software your project depends on, which is then automatically downloaded and installed during the build process
- Maven can build your project and generate archives, documentation, a project website, and other artifacts
- Maven supports unit testing, i.e., allows you to automatically check whether your code meets certain requirements

- Maven is maybe the most widely-used project build and dependency management tool in Java
- It allows you to specify which other software your project depends on, which is then automatically downloaded and installed during the build process
- Maven can build your project and generate archives, documentation, a project website, and other artifacts
- Maven supports unit testing, i.e., allows you to automatically check whether your code meets certain requirements
- Maven allows for automatic deployment (which we will not talk about here)

- Maven is maybe the most widely-used project build and dependency management tool in Java
- It allows you to specify which other software your project depends on, which is then automatically downloaded and installed during the build process
- Maven can build your project and generate archives, documentation, a project website, and other artifacts
- Maven supports unit testing, i.e., allows you to automatically check whether your code meets certain requirements
- Maven allows for automatic deployment (which we will not talk about here)
- Eclipse comes with Maven support

- Maven is maybe the most widely-used project build and dependency management tool in Java
- It allows you to specify which other software your project depends on, which is then automatically downloaded and installed during the build process
- Maven can build your project and generate archives, documentation, a project website, and other artifacts
- Maven supports unit testing, i.e., allows you to automatically check whether your code meets certain requirements
- Maven allows for automatic deployment (which we will not talk about here)
- Eclipse comes with Maven support
- Maven does not just define the project dependencies, but also the complete build process

- Maven is maybe the most widely-used project build and dependency management tool in Java
- It allows you to specify which other software your project depends on, which is then automatically downloaded and installed during the build process
- Maven can build your project and generate archives, documentation, a project website, and other artifacts
- Maven supports unit testing, i.e., allows you to automatically check whether your code meets certain requirements
- Maven allows for automatic deployment (which we will not talk about here)
- Eclipse comes with Maven support
- Maven does not just define the project dependencies, but also the complete build process
- Everything is versionized, so all builds are 100% reproducible (which sorts out the infamous “But it works on my machine. . .”)

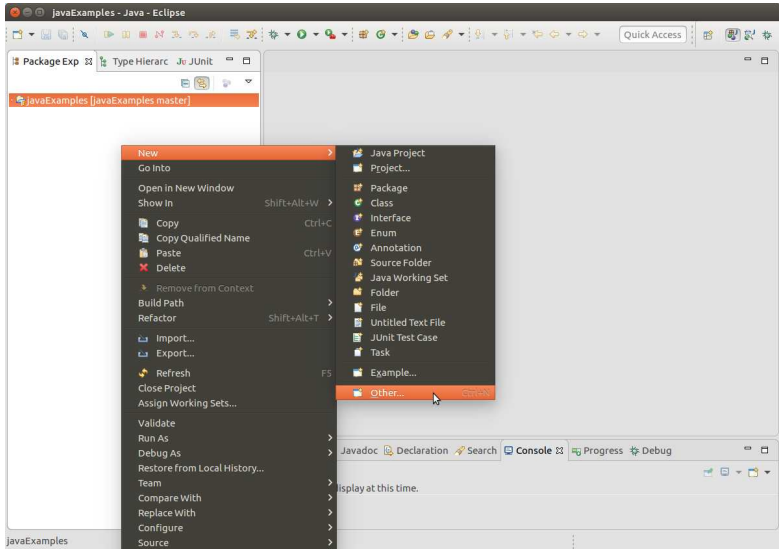


- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features

- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project

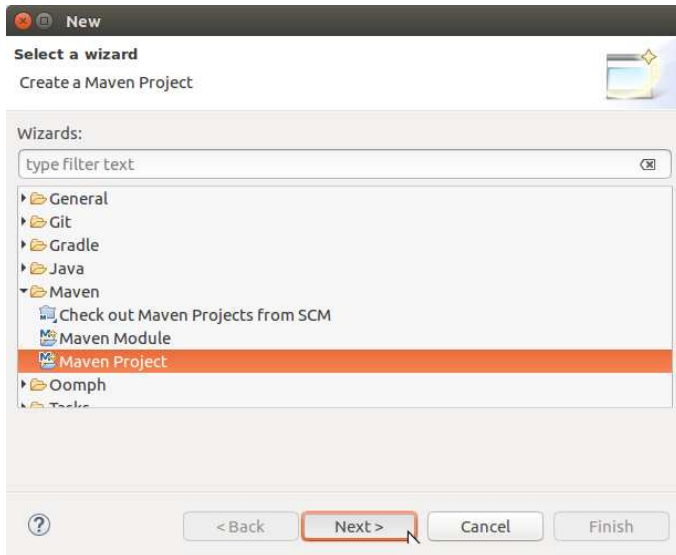
- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project:
  - First, we need to choose **New** and **Other...** from the **File** menu

# Creating and Building a Basic Maven Project



- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project:
  - First, we need to choose **New** and **Other...** from the **File** menu
  - In the next dialog, we open folder **Maven**, choose **Maven Project**, and click **Next**

# Creating and Building a Basic Maven Project



- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project:
  - First, we need to choose **New** and **Other...** from the **File** menu
  - In the next dialog, we open folder **Maven**, choose **Maven Project**, and click **Next**
  - We check **Create a simple project** and choose a suitable location via **Browse...**, then click **Next**

# Creating and Building a Basic Maven Project



**New Maven Project**

**New Maven project**  
Select project name and location

☒ Create a simple project (skip archetype selection)

☐ Use default workspace location

Location:

☐ Add project(s) to working set

Working set:

▶ Advanced



- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project:
  - First, we need to choose **New** and **Other...** from the **File** menu
  - In the next dialog, we open folder **Maven**, choose **Maven Project**, and click **Next**
  - We check **Create a simple project** and choose a suitable location via **Browse...**, then click **Next**
  - In the following form, we make the selections shown here (which I will explain later) and click **Finish**

- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project:
  - First, we need to choose **New** and **Other...** from the **File** menu
  - In the next dialog, we open folder **Maven**, choose **Maven Project**, and click **Next**
  - We check **Create a simple project** and choose a suitable location via **Browse...**, then click **Next**
  - In the following form, we make the selections shown here (which I will explain later) and click **Finish**:
    - Group Id: **cn.edu.hfu.iao**
    - Artifact Id: **simple-maven-project**
    - Version: **0.0.1**
    - Packaging: **jar**
    - Project name: **Simple Maven Project**

# Creating and Building a Basic Maven Project



**New Maven Project**

**New Maven project**  
Configure project

Artifact

Group Id:

Artifact Id:

Version:

Packaging:

Name:

Description:

Parent Project

Group Id:

Artifact Id:

Version:

Advanced

- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project:
  - First, we need to choose **New** and **Other...** from the **File** menu
  - In the next dialog, we open folder **Maven**, choose **Maven Project**, and click **Next**
  - We check **Create a simple project** and choose a suitable location via **Browse...**, then click **Next**
  - In the following form, we make the selections shown here (which I will explain later) and click **Finish**
  - A new Maven project has appeared, which basically is a special Eclipse Java project with a special folder structure and a file called **pom.xml** containing the project (and build) information

# Creating and Building a Basic Maven Project

A screenshot of the Eclipse IDE interface. The title bar reads 'javaExamples - Java - simple-maven-project/pom.xml - Eclipse'. The left sidebar shows the 'Package Explorer' with a tree view containing 'javaExamples [javaExamples master]', 'simple-maven-project [javaExamples master]', and its subdirectories: 'src/main/java', 'src/main/resources', 'src/test/java', 'src/test/resources', 'JRE System Library [J2SE-1.5]', '.settings', 'src', 'target', '.classpath', '.gitignore', '.project', and 'pom.xml' (which is selected and highlighted in orange). The main editor window displays the 'pom.xml' file with the following XML content:

```
1<?xml version="1.0" encoding="UTF-8"?>
2<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  <modelVersion>4.0.0</modelVersion>
4  <groupId>cn.edu.hfuu.iao</groupId>
5  <artifactId>simple-maven-project</artifactId>
6  <version>0.0.1</version>
7  <name>Simple Maven Project</name>
8  <description>A simple maven project without any advanced features.</description>
9</project>
```

The bottom of the IDE shows a tabbed interface with 'Overview', 'Dependencies', 'Dependency Hierarchy', 'Effective POM', and 'pom.xml' (selected). Below these tabs are sections for 'Problems', 'Javadoc', 'Declaration', 'Search', 'Console' (which is active and shows 'No consoles to display at this time.'), 'Progress', and 'Debug'.

## Listing: The contents of the Maven project file pom.xml

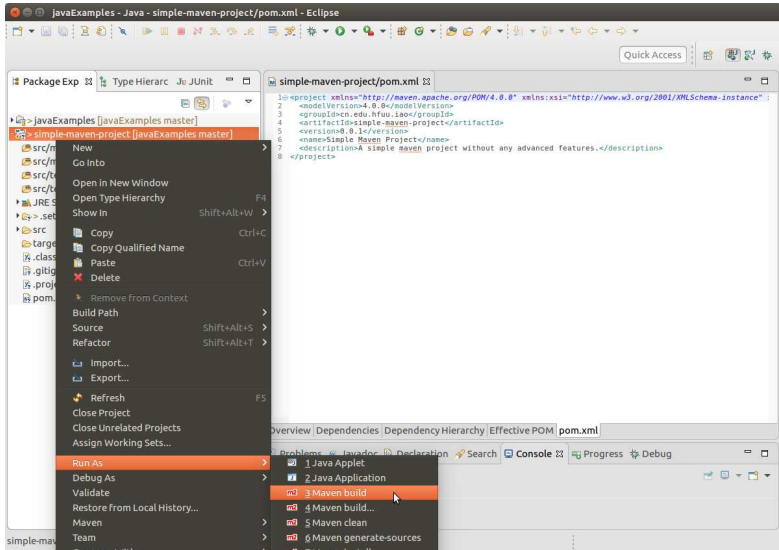
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>cn.edu.hfu.iao</groupId>
  <artifactId>simple-maven-project</artifactId>
  <version>0.0.1</version>
  <name>Simple Maven Project</name>
  <description>A simple maven project without any advanced
    features.</description>
</project>
```

- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project
- We can now build the project

- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project
- We can now build the project
  - right-click the project, choose `Run As` and then `Maven Build`

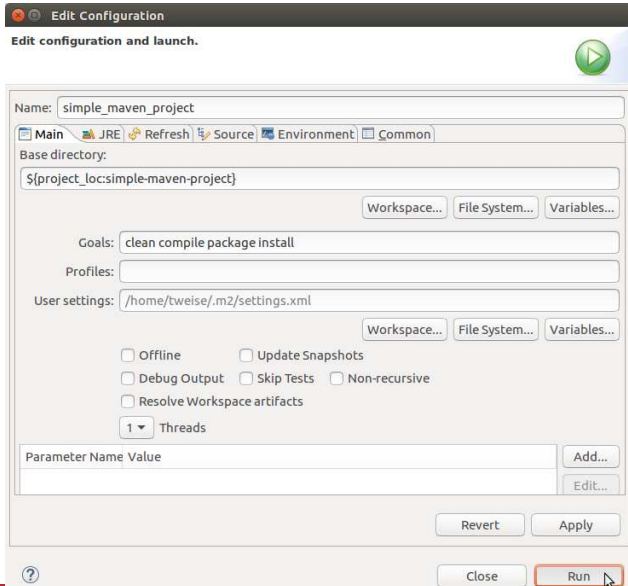


# Creating and Building a Basic Maven Project



- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project
- We can now build the project
  - right-click the project, choose `Run As` and then `Maven Build`
  - Under `Goals:` enter `clean compile package install` (just using `clean install` would do the same thing) then `Run`

# Creating and Building a Basic Maven Project



- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project
- We can now build the project
  - right-click the project, choose `Run As` and then `Maven Build`
  - Under `Goals:` enter `clean compile package install` (just using `clean install` would do the same thing) then `Run`
  - A build process will start which will first download several required modules, then compile your code (there is no code yet), and then creates a new `jar` archive (basically empty due to no code)

- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project
- We can now build the project
  - right-click the project, choose `Run As` and then `Maven Build`
  - Under `Goals:` enter `clean compile package install` (just using `clean install` would do the same thing) then `Run`
  - A build process will start which will first download several required modules, then compile your code (there is no code yet), and then creates a new `jar` archive (basically empty due to no code)
  - The generated *build artifacts* will be in folder `target`

# Creating and Building a Basic Maven Project



javaExamples - Java - simple-maven-project/pom.xml - Eclipse

Package Exp | Type Hierarc | JUnit |

javaExamples [javaExamples master]  
simple-maven-project [javaExamples master]  
src/main/java  
src/main/resources  
src/test/java  
src/test/resources  
JRE System Library [J2SE-1.5]  
settings  
src  
target  
maven-archiver  
maven-status  
simple-maven-project-0.0.1.jar  
classpath  
gitignore  
project  
pom.xml

simple-maven-project/pom.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<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>cn.edu.hfuu.iao</groupId>
  <artifactId>simple-maven-project</artifactId>
  <version>0.0.1</version>
  <name>Simple Maven Project</name>
  <description>A simple maven project without any advanced features.</description>
</project>
```

Overview | Dependencies | Dependency Hierarchy | Effective POM | pom.xml

Problems | Javadoc | Declaration | Search | Console | Progress | Debug

```
<terminated> simple_maven_project [Maven Build] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Feb 13, 2017, 6:04:3
WARNING) Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ simple-maven-project ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ simple-maven-project ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ simple-maven-project ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ simple-maven-project ---
[INFO] Skipping execution of surefire because it has already been run for this configuration
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ simple-maven-project ---
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ simple-maven-project ---
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-digest/1.0/plexus-digest-1.0
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-digest/1.0/plexus-digest-1.0
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-components/1.1.7/plexus-compo
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-components/1.1.7/plexus-compo
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-digest/1.0/plexus-digest-1.0
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-digest/1.0/plexus-digest-1.0
[INFO] Installing /home/tweise/local/programming/java/javaExamples/lessons/30/building with maven/simple_maven.p
[INFO] Installing /home/tweise/local/programming/java/javaExamples/lessons/30/building with maven/simple_maven.p
[INFO]
[INFO]
[INFO]
[INFO] Total time: 21.034 s
[INFO] Finished at: 2017-02-13T06:04:57+08:00
[INFO] Final Memory: 15M/296M
[INFO]
```

simple-maven-project-0.0.1.jar - simple-maven-project/target

- Before exploring the advanced features of Maven, let us start by creating and building a simple and plain Maven project and then discuss its basic features
- Creating a Simple Maven Project
- We can now build the project
  - right-click the project, choose `Run As` and then `Maven Build`
  - Under `Goals:` enter `clean compile package install` (just using `clean install` would do the same thing) then `Run`
  - A build process will start which will first download several required modules, then compile your code (there is no code yet), and then creates a new `jar` archive (basically empty due to no code)
  - The generated *build artifacts* will be in folder `target`
- Let us take a closer look on the stuff we just did

- The group ID identifies the “greater project”



- The group ID identifies the “greater project”
- It follows Java’s package naming convention

- The group ID identifies the “greater project”
- It follows Java’s package naming convention
- It has to at least identify a domain name **you control**

- The group ID identifies the “greater project”
- It follows Java’s package naming convention
- It has to at least identify a domain name **you control**
- In our case, this is `cn.edu.hfuu.iao`, because the *Institute of Applied Optimization* has domain `iao.hfuu.edu.cn`

- The group ID identifies the “greater project”
- It follows Java’s package naming convention
- It has to at least identify a domain name **you control**
- In our case, this is `cn.edu.hfuu.iao`, because the *Institute of Applied Optimization* has domain `iao.hfuu.edu.cn`
- It might have some additions for “greater projects”

- The group ID identifies the “greater project”
- It follows Java’s package naming convention
- It has to at least identify a domain name **you control**
- In our case, this is `cn.edu.hfuu.iao`, because the *Institute of Applied Optimization* has domain `iao.hfuu.edu.cn`
- It might have some additions for “greater projects”
- For instance, I could have used `cn.edu.hfuu.iao.teaching` as a group for all of our teaching projects

- The artifact ID is basically the name of the `jar` archive we want to generate without the version

- The artifact ID is basically the name of the `jar` archive we want to generate without the version
- You can consider it as a specially-formatted specific project name

- The artifact ID is basically the name of the `jar` archive we want to generate without the version
- You can consider it as a specially-formatted specific project name
- It is spelled in lower case letters and dashes are used ( `-` ) to separate name components



- The artifact ID is basically the name of the `jar` archive we want to generate without the version
- You can consider it as a specially-formatted specific project name
- It is spelled in lower case letters and dashes are used ( `-` ) to separate name components
- Basically, a group can contain several related projects with artifacts

- Every project always has a version

- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard

- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard, i.e., are of the form `major.minor.patch` version

- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard, i.e., are of the form `major.minor.patch` version
- Given a version number `major.minor.patch`, increment the

- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard, i.e., are of the form `major.minor.patch` version
- Given a version number `major.minor.patch`, increment the:
  - `major` version when you make incompatible API changes,

- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard, i.e., are of the form `major.minor.patch` version
- Given a version number `major.minor.patch`, increment the:
  - `major` version when you make incompatible API changes,
  - `minor` version when you add functionality in a backwards-compatible manner

- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard, i.e., are of the form `major.minor.patch` version
- Given a version number `major.minor.patch`, increment the:
  - `major` version when you make incompatible API changes,
  - `minor` version when you add functionality in a backwards-compatible manner, and
  - `patch` version when you make backwards-compatible bug fixes.



- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard, i.e., are of the form `major.minor.patch` version
- Given a version number `major.minor.patch`, increment the:
  - `major` version when you make incompatible API changes,
  - `minor` version when you add functionality in a backwards-compatible manner, and
  - `patch` version when you make backwards-compatible bug fixes.
- Code using a library of version `a1.b1.c1` will

- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard, i.e., are of the form `major.minor.patch` version
- Given a version number `major.minor.patch`, increment the:
  - `major` version when you make incompatible API changes,
  - `minor` version when you add functionality in a backwards-compatible manner, and
  - `patch` version when you make backwards-compatible bug fixes.
- Code using a library of version `a1.b1.c1` will
  - compile exactly the same with library version `a1.b1.c2`

- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard, i.e., are of the form `major.minor.patch` version
- Given a version number `major.minor.patch`, increment the:
  - `major` version when you make incompatible API changes,
  - `minor` version when you add functionality in a backwards-compatible manner, and
  - `patch` version when you make backwards-compatible bug fixes.
- Code using a library of version `a1.b1.c1` will
  - compile exactly the same with library version `a1.b1.c2`
  - compile exactly the same with library version `a1.b2.c2` if `b2`  $\geq$  `b1`

- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard, i.e., are of the form `major.minor.patch` version
- Given a version number `major.minor.patch`, increment the:
  - `major` version when you make incompatible API changes,
  - `minor` version when you add functionality in a backwards-compatible manner, and
  - `patch` version when you make backwards-compatible bug fixes.
- Code using a library of version `a1.b1.c1` will
  - compile exactly the same with library version `a1.b1.c2`
  - compile exactly the same with library version `a1.b2.c2` if `b2`  $\geq$  `b1`
  - may not compile with library version `a2.b2.c2` if `a1`  $\neq$  `a2`

- Every project always has a version
- The versions follow the **semantic versioning** (<http://semver.org/>) standard, i.e., are of the form `major.minor.patch` version
- Given a version number `major.minor.patch`, increment the:
  - `major` version when you make incompatible API changes,
  - `minor` version when you add functionality in a backwards-compatible manner, and
  - `patch` version when you make backwards-compatible bug fixes.
- Code using a library of version `a1.b1.c1` will
  - compile exactly the same with library version `a1.b1.c2`
  - compile exactly the same with library version `a1.b2.c2` if  $b2 \geq b1$
  - may not compile with library version `a2.b2.c2` if  $a1 \neq a2$
- Exception: If your project is still very experimental, you can use version `0.x.y`: For such versions, the rules for minor and patch level versions can be violated (but you should still try to not to)

- Maven prescribes a special project structure

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)



- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows:
  - `<root>` the project root folder

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows:
  - `<root>` the project root folder
  - `src` the folder for all source code

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows:
  - `<root>` the project root folder
  - `src` the folder for all source code
  - `target` generated during build: generated classes and artifacts

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows:
  - `<root>` the project root folder
  - `src` the folder for all source code
  - `target` generated during build: generated classes and artifacts
  - `pom.xml` the project settings

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows:

`<root>` the project root folder

`src` the folder for all source code

`main` the main folder: all program/library sources and resources

`target` generated during build: generated classes and artifacts

`pom.xml` the project settings

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows:

`<root>` the project root folder

`src` the folder for all source code

`main` the main folder: all program/library sources and resources

`test` the test folder: all test sources and resources

`target` generated during build: generated classes and artifacts

`pom.xml` the project settings

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows:

`<root>` the project root folder

`src` the folder for all source code

`main` the main folder: all program/library sources and resources

`java` the java source code / package hierarchy

`test` the test folder: all test sources and resources

`target` generated during build: generated classes and artifacts

`pom.xml` the project settings



- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows:

`<root>` the project root folder

`src` the folder for all source code

`main` the main folder: all program/library sources and resources

`java` the java source code / package hierarchy

`resources` resources (text, graphics, ...)

`test` the test folder: all test sources and resources

`target` generated during build: generated classes and artifacts

`pom.xml` the project settings

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows:

`<root>` the project root folder

`src` the folder for all source code

`main` the main folder: all program/library sources and resources

`java` the java source code / package hierarchy

`resources` resources (text, graphics, ...)

`test` the test folder: all test sources and resources

`java` the test java source code / package hierarchy

`target` generated during build: generated classes and artifacts

`pom.xml` the project settings

- Maven prescribes a special project structure
- It separates the Java source code files from resources such as text files or images
- It separates the actual program source code from the code for unit testing (similar to what we did in Lesson 27: *Testing with JUnit*)
- The structure is as follows:

`<root>` the project root folder

`src` the folder for all source code

`main` the main folder: all program/library sources and resources

`java` the java source code / package hierarchy

`resources` resources (text, graphics, ...)

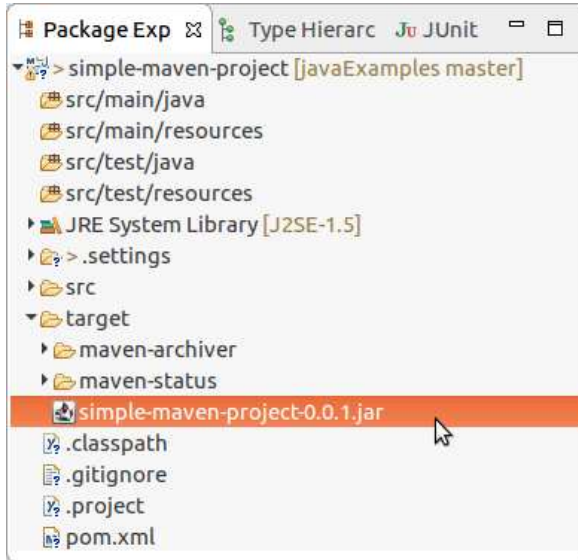
`test` the test folder: all test sources and resources

`java` the test java source code / package hierarchy

`resources` test resources (text, graphics, ...)

`target` generated during build: generated classes and artifacts

`pom.xml` the project settings



- The most important component of a Maven project is the `pom.xml` file

- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format

- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of `elements` and `attributes`

- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of `elements` and `attributes` in the form of

```
<elementName attribute1="value1" attribute2=...> ...element contents ...</elementName>
```



- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of `elements` and `attributes` in the form of

```
<elementName attribute1="value1" attribute2=...> ...element contents ...</elementName>
```
- It contains all the important information about the project

- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of `elements` and `attributes` in the form of

```
<elementName attribute1="value1" attribute2=...> ...element contents ...</elementName>
```
- It contains all the important information about the project, e.g.,
  - the basic infos (we use this in our simple project)

- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of `elements` and `attributes` in the form of

```
<elementName attribute1="value1" attribute2=...> ...element contents ...</elementName>
```
- It contains all the important information about the project, e.g.,
  - the basic infos (we use this in our simple project)
  - infos about the organization developing the project

- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of `elements` and `attributes` in the form of

```
<elementName attribute1="value1" attribute2=...> ...element contents ...</elementName>
```
- It contains all the important information about the project, e.g.,
  - the basic infos (we use this in our simple project)
  - infos about the organization developing the project
  - infos about the involved developers

- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of `elements` and `attributes` in the form of

```
<elementName attribute1="value1" attribute2=...> ...element contents ...</elementName>
```
- It contains all the important information about the project, e.g.,
  - the basic infos (we use this in our simple project)
  - infos about the organization developing the project
  - infos about the involved developers
  - property definitions to be used in the rest of the `pom`

- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of `elements` and `attributes` in the form of

```
<elementName attribute1="value1" attribute2=...> ...element contents ...</elementName>
```
- It contains all the important information about the project, e.g.,
  - the basic infos (we use this in our simple project)
  - infos about the organization developing the project
  - infos about the involved developers
  - property definitions to be used in the rest of the `pom`
  - license information

- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of `elements` and `attributes` in the form of

```
<elementName attribute1="value1" attribute2=...> ...element contents ...</elementName>
```
- It contains all the important information about the project, e.g.,
  - the basic infos (we use this in our simple project)
  - infos about the organization developing the project
  - infos about the involved developers
  - property definitions to be used in the rest of the `pom`
  - license information
  - infos about SCM, issue management, and the inception year

- The most important component of a Maven project is the pom.xml file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of elements and attributes in the form of

```
<elementName attribute1="value1" attribute2=...> ...element contents ...</elementName>
```
- It contains all the important information about the project, e.g.,
  - the basic infos (we use this in our simple project)
  - infos about the organization developing the project
  - infos about the involved developers
  - property definitions to be used in the rest of the pom
  - license information
  - infos about SCM, issue management, and the inception year
  - the dependencies (i.e., the libraries we need)



- The most important component of a Maven project is the `pom.xml` file
- This file is in the (much more general!) XML format, which prescribes a *hierarchical* structure of `elements` and `attributes` in the form of

```
<elementName attribute1="value1" attribute2=...> ...element contents ...</elementName>
```
- It contains all the important information about the project, e.g.,
  - the basic infos (we use this in our simple project)
  - infos about the organization developing the project
  - infos about the involved developers
  - property definitions to be used in the rest of the `pom`
  - license information
  - infos about SCM, issue management, and the inception year
  - the dependencies (i.e., the libraries we need)
  - the build process specification

## Listing: The contents of the Maven project file 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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>cn.edu.hfu.iao</groupId>
  <artifactId>simple-maven-project</artifactId>
  <version>0.0.1</version>
  <name>Simple Maven Project</name>
  <description>A simple maven project without any advanced
    features.</description>
</project>
```

- The Maven build process is not easy to understand

- The Maven build process is not easy to understand
- The build process consists of **phases**

- The Maven build process is not easy to understand
- The build process consists of **phases**, such as  
**clean** delete everything in the **target** folder

- The Maven build process is not easy to understand
- The build process consists of **phases**, such as
  - clean** delete everything in the **target** folder
  - validate** check whether the project is correct and all necessary info is there

- The Maven build process is not easy to understand
- The build process consists of **phases**, such as
  - clean** delete everything in the **target** folder
  - validate** check whether the project is correct and all necessary info is there
  - compile** compile the source code of the project

- The Maven build process is not easy to understand
- The build process consists of **phases**, such as
  - clean** delete everything in the **target** folder
  - validate** check whether the project is correct and all necessary info is there
  - compile** compile the source code of the project
  - test** run all tests, e.g., JUnit tests (fails if tests fail)



- The Maven build process is not easy to understand
- The build process consists of **phases**, such as
  - clean** delete everything in the **target** folder
  - validate** check whether the project is correct and all necessary info is there
  - compile** compile the source code of the project
  - test** run all tests, e.g., JUnit tests (fails if tests fail)
  - package** create the artifact package (in our case, the **jar** )

- The Maven build process is not easy to understand
- The build process consists of **phases**, such as
  - clean** delete everything in the `target` folder
  - validate** check whether the project is correct and all necessary info is there
  - compile** compile the source code of the project
  - test** run all tests, e.g., JUnit tests (fails if tests fail)
  - package** create the artifact package (in our case, the `jar` )
  - verify** run any checks on results of integration tests

- The Maven build process is not easy to understand
- The build process consists of **phases**, such as
  - clean** delete everything in the `target` folder
  - validate** check whether the project is correct and all necessary info is there
  - compile** compile the source code of the project
  - test** run all tests, e.g., JUnit tests (fails if tests fail)
  - package** create the artifact package (in our case, the `jar`)
  - verify** run any checks on results of integration tests
  - install** install the package into the local repository (for other builds depending on it)

- The Maven build process is not easy to understand
- The build process consists of **phases**, such as

**clean** delete everything in the `target` folder

**validate** check whether the project is correct and all necessary info is there

**compile** compile the source code of the project

**test** run all tests, e.g., JUnit tests (fails if tests fail)

**package** create the artifact package (in our case, the `jar`)

**verify** run any checks on results of integration tests

**install** install the package into the local repository (for other builds depending on it)

**deploy** release into environment

- The Maven build process is not easy to understand
- The build process consists of **phases**, such as
  - clean** delete everything in the `target` folder
  - validate** check whether the project is correct and all necessary info is there
  - compile** compile the source code of the project
  - test** run all tests, e.g., JUnit tests (fails if tests fail)
  - package** create the artifact package (in our case, the `jar`)
  - verify** run any checks on results of integration tests
  - install** install the package into the local repository (for other builds depending on it)
  - deploy** release into environment
- In Eclipse (or when using the Maven command line tool `mvn`), you only need to specify `clean` together the *last* phase to be executed and all phases leading up to it are executed

- An artifact is a result of the build process

- An artifact is a result of the build process
- Usually, this is an archive containing an executable, source, tests, or documentation

- An artifact is a result of the build process
- Usually, this is an archive containing an executable, source, tests, or documentation
- The name of an artifact is usually

```
artifactID-version[-classifier].<archiveType>
```



- An artifact is a result of the build process
- Usually, this is an archive containing an executable, source, tests, or documentation
- The name of an artifact is usually `artifactID-version[-classifier].<archiveType>`, where `artifactID` is the id of the project's main artifact, e.g., `simple-maven-project`

- An artifact is a result of the build process
- Usually, this is an archive containing an executable, source, tests, or documentation
- The name of an artifact is usually

`artifactID-version[-classifier].<archiveType>` , where

**artifactID** is the id of the project's main artifact, e.g., `simple-maven-project`

**version** is the version string, e.g., `0.0.1`

- An artifact is a result of the build process
- Usually, this is an archive containing an executable, source, tests, or documentation
- The name of an artifact is usually

`artifactID-version[-classifier].<archiveType>` , where

**artifactID** is the id of the project's main artifact, e.g., `simple-maven-project`

**version** is the version string, e.g., `0.0.1`

**[-classifier]** is an optional classifier for "side-artifacts"

- An artifact is a result of the build process
- Usually, this is an archive containing an executable, source, tests, or documentation
- The name of an artifact is usually

`artifactID-version[-classifier].<archiveType>` , where

**artifactID** is the id of the project's main artifact, e.g., `simple-maven-project`

**version** is the version string, e.g., `0.0.1`

**[-classifier]** is an optional classifier for "side-artifacts", such as

- `-src` for archives containing the source code and resources (*not* the generated `.class` files)

- An artifact is a result of the build process
- Usually, this is an archive containing an executable, source, tests, or documentation
- The name of an artifact is usually

`artifactID-version[-classifier].<archiveType>` , where

`artifactID` is the id of the project's main artifact, e.g., `simple-maven-project`

`version` is the version string, e.g., `0.0.1`

`[-classifier]` is an optional classifier for "side-artifacts", such as

- `-src` for archives containing the source code and resources (*not* the generated `.class` files)
- `-javadoc` for archives containing the generated Javadoc documentation

- An artifact is a result of the build process
- Usually, this is an archive containing an executable, source, tests, or documentation
- The name of an artifact is usually

`artifactID-version[-classifier].<archiveType>` , where

**artifactID** is the id of the project's main artifact, e.g., `simple-maven-project`

**version** is the version string, e.g., `0.0.1`

**[-classifier]** is an optional classifier for "side-artifacts", such as

- `-src` for archives containing the source code and resources (*not* the generated `.class` files)
- `-javadoc` for archives containing the generated Javadoc documentation
- `-tests` for generating the compiled tests

- An artifact is a result of the build process
- Usually, this is an archive containing an executable, source, tests, or documentation
- The name of an artifact is usually

`artifactID-version[-classifier].<archiveType>` , where

**artifactID** is the id of the project's main artifact, e.g., `simple-maven-project`

**version** is the version string, e.g., `0.0.1`

**[-classifier]** is an optional classifier for "side-artifacts", such as

- `-src` for archives containing the source code and resources (*not* the generated `.class` files)
- `-javadoc` for archives containing the generated Javadoc documentation
- `-tests` for generating the compiled tests

**archiveType** is usually `jar` , but for web projects it may be stuff like `war` , `aar` , `ear` , which are all "special" `jar` archives

- An artifact is a result of the build process
- Usually, this is an archive containing an executable, source, tests, or documentation
- The name of an artifact is usually

`artifactID-version[-classifier].<archiveType>`, where

**artifactID** is the id of the project's main artifact, e.g., `simple-maven-project`

**version** is the version string, e.g., `0.0.1`

**[-classifier]** is an optional classifier for "side-artifacts", such as

- `-src` for archives containing the source code and resources (*not* the generated `.class` files)
- `-javadoc` for archives containing the generated Javadoc documentation
- `-tests` for generating the compiled tests

**archiveType** is usually `jar`, but for web projects it may be stuff like `war`, `aar`, `ear`, which are all "special" `jar` archives

- Our simple project generated artifact `simple-maven-project-0.0.1.jar`



- Let us now create a more advanced Maven project

- Let us now create a more advanced Maven project, which
  - provides more information about our team and tools

- Let us now create a more advanced Maven project, which
  - provides more information about our team and tools,
  - depends on another library (commons math 3 from Apache)

- Let us now create a more advanced Maven project, which
  - provides more information about our team and tools,
  - depends on another library (commons math 3 from Apache),
  - produces an executable `jar` archive

- Let us now create a more advanced Maven project, which
  - provides more information about our team and tools,
  - depends on another library (commons math 3 from Apache),
  - produces an executable `jar` archive, and
  - generates Javadoc (in an archive)

- Let us now create a more advanced Maven project, which
  - provides more information about our team and tools,
  - depends on another library (commons math 3 from Apache),
  - produces an executable `jar` archive, and
  - generates Javadoc (in an archive)
- For this purpose, we generate a simple Maven project in exactly the same way as before, but name the artifact `project-with-dependencies`

- We then edit the generated `pom.xml` file to look as follows

## Listing: The contents of the pom.xml after we edit it

```

<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/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>cn.edu.hfuu.iao</groupId>
  <artifactId>project-with-dependencies</artifactId>
  <version>0.0.1</version>
  <name>Project with Dependencies</name>
  <description>A Maven project with dependencies and
    more information.
    also generating an executable JAR and
    Javadoc.</description>

  <url>http://iao.hfuu.edu.cn/</url>
  <organization>
    <url>http://iao.hfuu.edu.cn/</url>
    <name>Institute of Applied Optimization (IAO)</name>
  </organization>

  <developers>
    <developer>
      <id>thomasweise</id>
      <name>Thomas Weise</name>
      <email>weise@hfuu.edu.cn</email>
      <url>http://iao.hfuu.edu.cn/index.php/team/director/</url>
      <organization>Institute of Applied Optimization
        (IAO)</organization>
      <organizationUrl>http://iao.hfuu.edu.cn/</organizationUrl>
      <roles>
        <role>architect</role>
        <role>developer</role>
        <role>
          <timezones>China Time Zone</timezones>
        </role>
      </roles>
    </developer>
  </developers>

  <properties>
    <encoding>UTF-8</encoding>
    <project.build.sourceEncoding>${encoding}</project.build.sourceEncoding>
    <project.reporting.outputEncoding>${encoding}</project.reporting.outputEncoding>
    <jdk.version>1.8</jdk.version>
  </properties>

  <licenses>
    <license>
      <name>GNU GENERAL PUBLIC LICENSE Version 3, 29
        June 2007</name>
      <url>http://www.gnu.org/licenses/gpl-3.0-standalone.html</url>
      <distribution>repo</distribution>
    </license>
  </licenses>

  <issueManagement>
    <url>https://github.com/thomasweise/javaExamples/issues</url>
    <system>GitHub</system>
  </issueManagement>

  <scm>
    <connection>scm:git:git@github.com:thomasweise/javaExamples.git</connection>
    <developerConnection>
      scm:git:git@github.com:thomasweise/javaExamples.git</developerConnection>
    <url>git@github.com:thomasweise/javaExamples.git</url>
  </scm>

  <inceptionYear>2017</inceptionYear>

  <dependencies>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-math3</artifactId>
      <version>3.6.1</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>${jdk.version}</source>
          <target>${jdk.version}</target>
          <encoding>${encoding}</encoding>
          <showWarnings>true</showWarnings>
          <showDeprecation>true</showDeprecation>
        </configuration>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>2.9.1</version>
        <configuration>
          <show>private</show>
          <detectLinks>true</detectLinks>
          <detectJavaApiLink>true</detectJavaApiLink>
          <quiet>true</quiet>
        </configuration>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.6</version>
        <configuration>
          <archive>
            <manifest>
              <addDefaultImplementationEntries />
              <addDefaultSpecificationEntries />
            </manifest>
          </archive>
        </configuration>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>3.0</version>
        <configuration>
          <goals>
            <goal>attached</goal>
          </goals>
          <phase>package</phase>
        </configuration>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <includePom>true</includePom>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <includePom>true</includePom>
  <useDefaultExcludes>true</useDefaultExcludes>
  <useDefaultManifestFile>false</useDefaultManifestFile>
  </configuration>
  <execution>
    <id>attach-sources</id>
    <goals>
      <goal>jar</goal>
    </goals>
    </execution>
  </executions>
  </plugin>

  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.6</version>
    <configuration>
      <archive>
        <manifest>
          <addDefaultImplementationEntries />
          <addDefaultSpecificationEntries />
        </manifest>
      </archive>
    </configuration>
  </plugin>

  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>3.0</version>
    <configuration>
      <goals>
        <goal>attached</goal>
      </goals>
      <phase>package</phase>
    </configuration>
  </plugin>

  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-source-plugin</artifactId>
    <version>2.3</version>
    <configuration>
      <includePom>true</includePom>
    </configuration>
  </plugin>
</project>

```



- We then edit the generated `pom.xml` file to look as follows
- OK, thank you, now the details

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)

## Listing: pom.xml Lines 6–12: Basic Info

```
<modelVersion>4.0.0</modelVersion>
<groupId>cn.edu.hfuu.iao</groupId>
<artifactId>project-with-dependencies</artifactId>
<version>0.0.1</version>
<name>Project with Dependencies</name>
<description>A Maven project with dependencies and
    more information,
also generating an executable JAR and
    Javadoc.</description>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project

## Listing: pom.xml Lines 15–19: Organization Info

```
<url>http://iao.hfuu.edu.cn/</url>
<organization>
  <url>http://iao.hfuu.edu.cn/</url>
  <name>Institute of Applied Optimization (IAO)</name>
</organization>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project

## Listing: pom.xml Lines 21–35: Developer Info

```
<developers>
  <developer>
    <id>thomasWeise</id>
    <name>Thomas Weise</name>
    <email>tweise@hfuu.edu.cn</email>
    <url>http://iao.hfuu.edu.cn/index.php/team/director/</url>
    <organization>Institute of Applied Optimization
      (IAO)</organization>
    <organizationUrl>http://iao.hfuu.edu.cn/</organizationUrl>
    <roles>
      <role>architect</role>
      <role>developer</role>
    </roles>
    <timezone>China Time Zone</timezone>
  </developer>
</developers>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties: contents of element `<n>contents</n>` become available as `${n}`



## Listing: pom.xml Lines 37–42: Properties

```
<properties>
  <encoding>UTF-8</encoding>
  <project.build.sourceEncoding>${encoding}</project.build.sourceEncoding>
  <project.reporting.outputEncoding>${encoding}</project.reporting.outputEncoding>
  <jdk.version>1.8</jdk.version>
</properties>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties: contents of element `<n>contents</n>` become available as `${n}`
  - Licensing Information: here GPL version 3

## Listing: pom.xml Lines 44–50: License

```
<licenses>
  <license>
    <name>GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007</name>
    <url>http://www.gnu.org/licenses/gpl-3.0-standalone.html</url>
    <distribution>repo</distribution>
  </license>
</licenses>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties: contents of element `<n>contents</n>` become available as `${n}`
  - Licensing Information: here GPL version 3
  - Issue management: where to report errors

## Listing: pom.xml Lines 52–55: Issue Management

```
<issueManagement>  
  <url>https://github.com/thomasWeise/javaExamples/issues</url>  
  <system>GitHub</system>  
</issueManagement>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties: contents of element `<n>contents</n>` become available as `${n}`
  - Licensing Information: here GPL version 3
  - Issue management: where to report errors
  - Software configuration management: here our `git` repository

## Listing: pom.xml Lines 57–62: Software Configuration Management

```
<scm>
  <connection>scm:git:git@github.com:thomasWeise/javaExamples.git</connection>
  <developerConnection>
    scm:git:git@github.com:thomasWeise/javaExamples.git</developerConnection>
  <url>git@github.com:thomasWeise/javaExamples.git</url>
</scm>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties
  - Licensing Information: here GPL version 3
  - Issue management: where to report errors
  - Software configuration management: here our `git` repository
  - Inception year: when did the project start



Listing: `pom.xml` Lines 64: Inception Year

```
<inceptionYear>2017</inceptionYear>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties
  - Licensing Information: here GPL version 3
  - Issue management: where to report errors
  - Software configuration management: here our `git` repository
  - Inception year: when did the project start
  - Dependencies: Which other libraries does our project need?

## Listing: pom.xml Lines 67–73: Dependencies

```
<dependencies>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-math3</artifactId>
    <version>3.6.1</version>
  </dependency>
</dependencies>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties
  - Licensing Information: here GPL version 3
  - Issue management: where to report errors
  - Software configuration management: here our `git` repository
  - Inception year: when did the project start
  - Dependencies: Which other libraries does our project need?
  - Compilation process: here using Java `${jdk.version}` which was set to 1.8

## Listing: pom.xml Lines 75–89: Build (1): Compilation

```
<build>

  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>${jdk.version}</source>
        <target>${jdk.version}</target>
        <encoding>${encoding}</encoding>
        <showWarnings>true</showWarnings>
        <showDeprecation>true</showDeprecation>
      </configuration>
    </plugin>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties
  - Licensing Information: here GPL version 3
  - Issue management: where to report errors
  - Software configuration management: here our `git` repository
  - Inception year: when did the project start
  - Dependencies: Which other libraries does our project need?
  - Compilation process: here using Java `${jdk.version}` which was set to 1.8
- We also want a `jar` containing the generated Javadoc

## Listing: pom.xml Lines 91–109: Build (2): Javadoc jar

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-javadoc-plugin</artifactId>
  <version>2.9.1</version>
  <configuration>
    <show>private</show>
    <detectLinks>true</detectLinks>
    <detectJavaApiLink>true</detectJavaApiLink>
    <quiet>true</quiet>
  </configuration>
  <executions>
    <execution>
      <id>attach-javadoc</id>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties
  - Licensing Information: here GPL version 3
  - Issue management: where to report errors
  - Software configuration management: here our `git` repository
  - Inception year: when did the project start
  - Dependencies: Which other libraries does our project need?
  - Compilation process: here using Java `${jdk.version}` which was set to 1.8
  - We also want a `jar` containing the generated Javadoc
  - We also want a `jar` containing all the source code



## Listing: pom.xml Lines 111–128: Build (3): Sources jar

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-source-plugin</artifactId>
  <version>2.3</version>
  <configuration>
    <includePom>true</includePom>
    <useDefaultExcludes>true</useDefaultExcludes>
    <useDefaultManifestFile>false</useDefaultManifestFile>
  </configuration>
  <executions>
    <execution>
      <id>attach-sources</id>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties
  - Licensing Information: here GPL version 3
  - Issue management: where to report errors
  - Software configuration management: here our `git` repository
  - Inception year: when did the project start
  - Dependencies: Which other libraries does our project need?
  - Compilation process: here using Java `${jdk.version}` which was set to 1.8
  - We also want a `jar` containing the generated Javadoc
  - We also want a `jar` containing all the source code
  - Generate the “main” artifact: an executable `jar` with main class `cn.edu.hfu.iao.Main`

## Listing: pom.xml Lines 131–144: Build (4): (executable) jar

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <archive>
      <manifest>
        <addDefaultImplementationEntries />
        <addDefaultSpecificationEntries />
        <mainClass>cn.edu.hfu.iao.Main</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties
  - Licensing Information: here GPL version 3
  - Issue management: where to report errors
  - Software configuration management: here our `git` repository
  - Inception year: when did the project start
  - Dependencies: Which other libraries does our project need?
  - Compilation process: here using Java `${jdk.version}` which was set to 1.8
  - We also want a `jar` containing the generated Javadoc
  - We also want a `jar` containing all the source code
  - Generate the “main” artifact
  - Generate an executable `jar` *including* all dependencies (here `commons-math3` ) with main class `cn.edu.hfu.iao.Main`

## Listing: pom.xml Lines 146–172: Build (5): (executable) jar with dependencies

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>attached</goal>
      </goals>
      <phase>package</phase>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
          <manifest>
            <mainClass>cn.edu.hfu.iao.Main</mainClass>
          </manifest>
        </archive>
      </configuration>
    </execution>
  </executions>
</plugin>

</plugins>
</build>

</project>
```

- We then edit the generated `pom.xml` file to look as follows
  - Basic Project Information (almost the same as before)
  - Project URL and info about the organization behind project
  - The developers working on the project
  - Properties
  - Licensing Information: here GPL version 3
  - Issue management: where to report errors
  - Software configuration management: here our `git` repository
  - Inception year: when did the project start
  - Dependencies: Which other libraries does our project need?
  - Compilation process: here using Java `${jdk.version}` which was set to 1.8
  - We also want a `jar` containing the generated Javadoc
  - We also want a `jar` containing all the source code
  - Generate the “main” artifact
  - Generate an executable `jar` *including* all dependencies
- The `pom.xml` specifies quite a complex build process!

## Listing: Our Main class using Dependency

```

package cn.edu.hfuu.iao;

import java.util.Scanner;

import org.apache.commons.math3.stat.regression.SimpleRegression; // import class from dependency library

/** The main class of our project: it reads data from stdin and returns a linear function fitting to it */
public class Main {

    /** read data from a Scanner, return a SimpleRegression instance with the fitting result */
    static final SimpleRegression fitLine(final Scanner scanner) {
        SimpleRegression regression = new SimpleRegression(); // using commons-math3's simple regression class
        for (;;) {
            if (!scanner.hasNextDouble()) { break; } // if there is no double number, stop reading
            double x = scanner.nextDouble(); // ok, there is one, read it as x coordinate
            if (!scanner.hasNextDouble()) { break; } // if there is no double number, stop reading
            double y = scanner.nextDouble(); // ok, there is one, read it as y coordinate
            regression.addData(x, y); // add the new x and y coordinate to the dataset
        }
        return regression;
    }

    /** The main routine
     * @param args
     * we ignore this parameter for now */
    public static final void main(final String[] args) {
        System.err.println("Welcome to the linear curve fitting program."); // $NON-NLS-1$
        System.err.println("Enter point pairs one pair a line, x and y coordinates separated by space or tab."); // $NON-NLS-1$
        System.err.println("Linear curve is fitted when stdin ends or Ctrl-D is pressed."); // $NON-NLS-1$

        SimpleRegression regression; // using commons-math3's simple regression class
        try (final Scanner scanner = new Scanner(System.in)) { // using a Scanner in try-with-resource on System.in
            regression = fitLine(scanner); // fit the data from the scanner
        }

        System.out.print("y\u2248"); // $NON-NLS-1$ // print "y is approximately "
        System.out.print(regression.getIntercept()); // print the y coordinate at x=0
        System.out.print(", x\u2248"); // $NON-NLS-1$ // " + x + "
        System.out.print(regression.getSlope()); // print the slope
        System.out.print("(root mean square error: "); // print RMSE: is 0 if data is linear // $NON-NLS-1$
        System.out.print(Math.sqrt(regression.getMeanSquareError()));
        System.out.println(')');
    }
}

```

- We now can build our project using goals `clean install`



JavaExamples - Java - project-with-dependencies/pom.xml - Eclipse

Package Explorer | Type Hierarchy | JUnit

project-with-dependencies [JavaExamples master]

- src/main/java
  - cn.edu.hfu.iao
    - Main.java
    - package-info.java
- src/main/resources
- src/test/java
- src/test/resources
- JRE System Library [JavaSE-1.8]
- Maven Dependencies
- .settings
- src
- target
  - apidocs
  - archive-tmp
  - generated-sources
  - javadoc-bundle-options
  - maven-archiver
  - maven-status
  - project-with-dependencies-0.0.1-jar-with-dependencies.jar
  - project-with-dependencies-0.0.1-javadoc.jar
  - project-with-dependencies-0.0.1-sources.jar
  - project-with-dependencies-0.0.1.jar
  - .classpath
  - .gitignore
  - .project
  - pom.xml

4 items selected

project-with-dependencies/pom.xml | package-info.java | Main.java

```

138     <addDefaultImplementationEntries />
139     <addDefaultSpecificationEntries />
140     <mainClass>cn.edu.hfu.iao.Main</mainClass>
141   </manifest>
142   </archive>
  .. </configuration>
    
```

Overview | Dependencies | Dependency Hierarchy | Effective POM | pom.xml

Problems | Javadoc | Declaration | Search | Console | Progress | Debug

```

<terminated> project_with_dependencies (2) [Maven Build] /usr/lib/jvm/java-8-openjdk-amd64/bin/java
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ project-with-dependencies ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ project-with-dependencies ---
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ project-with-dependencies ---
[INFO] Building jar: /home/tweise/local/programming/java/javaExamples/lessons/30_building_with_maven/
[INFO]
[INFO] --- maven-javadoc-plugin:2.9.1:jar (attach-javadoc) @ project-with-dependencies ---
[INFO] Building jar: /home/tweise/local/programming/java/javaExamples/lessons/30_building_with_maven/
[INFO]
[INFO] >>> maven-source-plugin:2.3:jar (attach-sources) > generate-sources @ project-with-dependencies
[INFO]
[INFO] <<< maven-source-plugin:2.3:jar (attach-sources) < generate-sources @ project-with-dependencies
[INFO]
[INFO] --- maven-source-plugin:2.3:jar (attach-sources) @ project-with-dependencies ---
[INFO] Building jar: /home/tweise/local/programming/java/javaExamples/lessons/30_building_with_maven/
[INFO]
[INFO] --- maven-assembly-plugin:2.2-beta-5:attached (default) @ project-with-dependencies ---
[INFO] Building jar: /home/tweise/local/programming/java/javaExamples/lessons/30_building_with_maven/
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ project-with-dependencies ---
[INFO] Installing /home/tweise/local/programming/java/javaExamples/lessons/30_building_with_maven/prc
[INFO] Installing /home/tweise/local/programming/java/javaExamples/lessons/30_building_with_maven/prc
[INFO] Installing /home/tweise/local/programming/java/javaExamples/lessons/30_building_with_maven/prc
[INFO] Installing /home/tweise/local/programming/java/javaExamples/lessons/30_building_with_maven/prc
[INFO] Installing /home/tweise/local/programming/java/javaExamples/lessons/30_building_with_maven/prc
[INFO] Installing /home/tweise/local/programming/java/javaExamples/lessons/30_building_with_maven/prc
    
```

- We now can build our project using goals `clean install`
- We find that the `target` folder now contains several artifacts

- We now can build our project using goals `clean install`
- We find that the `target` folder now contains several artifacts, namely
  - `project-with-dependencies-0.0.1.jar` – the main, executable `jar` of our project; requires `commons-math3` in the classpath to run

- We now can build our project using goals `clean install`
- We find that the `target` folder now contains several artifacts, namely
  - `project-with-dependencies-0.0.1.jar` – the main, executable `jar` of our project; requires `commons-math3` in the classpath to run
  - `project-with-dependencies-0.0.1-jar-with-dependencies.jar` – an executable `jar` of our project; contains `commons-math3`, can run directly

- We now can build our project using goals `clean install`
- We find that the `target` folder now contains several artifacts, namely
  - `project-with-dependencies-0.0.1.jar` – the main, executable `jar` of our project; requires `commons-math3` in the classpath to run
  - `project-with-dependencies-0.0.1-jar-with-dependencies.jar` – an executable `jar` of our project; contains `commons-math3`, can run directly
  - `project-with-dependencies-0.0.1-javadoc.jar` : the generated Javadoc documentation of our project (remember, a `jar` is basically a `zip` archive...)

- We now can build our project using goals `clean install`
- We find that the `target` folder now contains several artifacts, namely
  - `project-with-dependencies-0.0.1.jar` – the main, executable `jar` of our project; requires `commons-math3` in the classpath to run
  - `project-with-dependencies-0.0.1-jar-with-dependencies.jar` – an executable `jar` of our project; contains `commons-math3`, can run directly
  - `project-with-dependencies-0.0.1-javadoc.jar` : the generated Javadoc documentation of our project (remember, a `jar` is basically a `zip` archive...)
  - `project-with-dependencies-0.0.1-sources.jar` : a `jar` archive containing the source code of our project (easy for distribution)

- You have maybe noticed that our project depends on Apache commons-math3

- You have maybe noticed that our project depends on Apache commons-math3
- But we never downloaded that library



- You have maybe noticed that our project depends on Apache commons-math3
- But we never downloaded that library, yet it is in the classpath in Eclipse, and even inside one of our `jar` archives

- You have maybe noticed that our project depends on Apache commons-math3
- But we never downloaded that library, yet it is in the classpath in Eclipse, and even inside one of our `jar` archives
- Maven automatically downloaded it for us

- You have maybe noticed that our project depends on Apache commons-math3
- But we never downloaded that library, yet it is in the classpath in Eclipse, and even inside one of our `jar` archives
- Maven automatically downloaded it for us
- Maven uses *repositories*

- You have maybe noticed that our project depends on Apache commons-math3
- But we never downloaded that library, yet it is in the classpath in Eclipse, and even inside one of our `jar` archives
- Maven automatically downloaded it for us
- Maven uses *repositories*:
  - A repository is basically a special directory structure based on group IDs, artifact IDs, and (semantic) versions

- You have maybe noticed that our project depends on Apache commons-math3
- But we never downloaded that library, yet it is in the classpath in Eclipse, and even inside one of our `jar` archives
- Maven automatically downloaded it for us
- Maven uses *repositories*:
  - A repository is basically a special directory structure based on group IDs, artifact IDs, and (semantic) versions
  - For each such “coordinates”, we can determine a folder where the `jar` artifacts (library, source, javadoc) should be located

- You have maybe noticed that our project depends on Apache commons-math3
- But we never downloaded that library, yet it is in the classpath in Eclipse, and even inside one of our `jar` archives
- Maven automatically downloaded it for us
- Maven uses *repositories*:
  - A repository is basically a special directory structure based on group IDs, artifact IDs, and (semantic) versions
  - For each such “coordinates”, we can determine a folder where the `jar` artifacts (library, source, javadoc) should be located
  - There is one central repository in the internet, where organizations can register themselves and upload their open source artifacts

- You have maybe noticed that our project depends on Apache commons-math3
- But we never downloaded that library, yet it is in the classpath in Eclipse, and even inside one of our `jar` archives
- Maven automatically downloaded it for us
- Maven uses *repositories*:
  - A repository is basically a special directory structure based on group IDs, artifact IDs, and (semantic) versions
  - For each such “coordinates”, we can determine a folder where the `jar` artifacts (library, source, javadoc) should be located
  - There is one central repository in the internet, where organizations can register themselves and upload their open source artifacts
  - Whenever you need one of these public libraries (via your dependencies), Maven can find it and download it automatically

- You have maybe noticed that our project depends on Apache commons-math3
- But we never downloaded that library, yet it is in the classpath in Eclipse, and even inside one of our `jar` archives
- Maven automatically downloaded it for us
- Maven uses *repositories*:
  - A repository is basically a special directory structure based on group IDs, artifact IDs, and (semantic) versions
  - For each such “coordinates”, we can determine a folder where the `jar` artifacts (library, source, javadoc) should be located
  - There is one central repository in the internet, where organizations can register themselves and upload their open source artifacts
  - Whenever you need one of these public libraries (via your dependencies), Maven can find it and download it automatically
  - There also is a “local” repository on your machine, where dependencies are cached (and your compiled artifacts are `install` ed into)



- We now want to investigate the JUnit integration in Maven

- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before

- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`

- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`
- We make the following changes to the Maven `pom.xml` file

- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`
- We make the following changes to the Maven `pom.xml` file:
  - We adapt the basic project information to fit to the new project name

## Listing: pom.xml Lines 5–11: Basic Info

```
<modelVersion>4.0.0</modelVersion>
<groupId>cn.edu.hfuu.iao</groupId>
<artifactId>project-with-tests</artifactId>
<version>0.0.1</version>
<name>Project with Tests</name>
<description>A project similar to "Project with
    Dependencies",
but now also performing JUnit tests.</description>
```

- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`
- We make the following changes to the Maven `pom.xml` file:
  - We adapt the basic project information to fit to the new project name
  - We add a dependency on `JUnit`

- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`
- We make the following changes to the Maven `pom.xml` file:
  - We adapt the basic project information to fit to the new project name
  - We add a dependency on `JUnit`, but different from the `commons-math3` dependency, it gets scope “test”



- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`
- We make the following changes to the Maven `pom.xml` file:
  - We adapt the basic project information to fit to the new project name
  - We add a dependency on `JUnit`, but different from the `commons-math3` dependency, it gets scope “test”, since it is only needed during compilation and testing

- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`
- We make the following changes to the Maven `pom.xml` file:
  - We adapt the basic project information to fit to the new project name
  - We add a dependency on `JUnit`, but different from the `commons-math3` dependency, it gets scope “test”, since it is only needed during compilation and testing and not part of the final application (or “jar-with-dependencies”-jar)

## Listing: pom.xml Lines 66–78: Added Test-time Dependency on JUnit

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-math3</artifactId>
    <version>3.6.1</version>
  </dependency>
</dependencies>
```

- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`
- We make the following changes to the Maven `pom.xml` file:
  - We adapt the basic project information to fit to the new project name
  - We add a dependency on `JUnit`
  - We use the `surefire` plugin in the build process while will execute the JUnit tests for us

Listing: pom.xml Lines 173–177: Using surefire Plugin (runs tests)

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.18</version>
</plugin>
```

- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`
- We make the following changes to the Maven `pom.xml` file:
  - We adapt the basic project information to fit to the new project name
  - We add a dependency on `JUnit`
  - We use the `surefire` plugin in the build process while will execute the JUnit tests for us
  - We can also generate a nice HTML report about the whole project and the test results

- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`
- We make the following changes to the Maven `pom.xml` file:
  - We adapt the basic project information to fit to the new project name
  - We add a dependency on `JUnit`
  - We use the `surefire` plugin in the build process while will execute the JUnit tests for us
  - We can also generate a nice HTML report about the whole project and the test results, for this purpose we invoke the `site` goal when building and add a set of reporting plugins the the `pom.xml`

## Listing: pom.xml Lines 182–195: Using surefire Plugin Report HTML

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <version>2.7</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-report-plugin</artifactId>
      <version>2.18</version>
    </plugin>
  </plugins>
</reporting>
```



- We now want to investigate the JUnit integration in Maven
- For this purpose, we create a new Maven project in Eclipse with an artifact called `project-with-tests` in the same way as before
- We can basically copy everything from `project-with-dependencies` into the new project, even the Maven `pom.xml`
- We make the following changes to the Maven `pom.xml` file:
  - We adapt the basic project information to fit to the new project name
  - We add a dependency on `JUnit`
  - We use the `surefire` plugin in the build process while will execute the JUnit tests for us
  - We can also generate a nice HTML report about the whole project and the test results, for this purpose we invoke the `site` goal when building and add a set of reporting plugins the the `pom.xml`
- In the next step, we can add a JUnit test and place it into the same package as the real code, just in the `src/test/java` hierarchy

## Listing: The JUnit test of our Main class

```
package cn.edu.hfuu.iao;

import java.io.StringReader;
import java.util.Scanner;

import org.apache.commons.math3.stat.regression.SimpleRegression;
import org.junit.Assert;
import org.junit.Test;

/** The unit test for our line-fitting main routine */
public class MainTest {

    /** test whether (0,0) and (1,1) will fit to  $y=0+x*1$  */
    @Test
    public void testFitting0011ResultsIn01() {
        SimpleRegression regression;

        try (final StringReader sr = new StringReader("0_0\n1_1")) { //$NON-NLS-1$
            try (final Scanner scanner = new Scanner(sr)) {
                regression = Main.fitLine(scanner);
            }
        }

        Assert.assertEquals(0d, regression.getIntercept(), 1e-10d);
        Assert.assertEquals(1d, regression.getSlope(), 1e-10d);
    }
}
```

- We can now build the new project by invoking Maven with the goals  
`clean test install site`

- We can now build the new project by invoking Maven with the goals  
`clean test install site`
- The build will compile our code

- We can now build the new project by invoking Maven with the goals  
`clean test install site`
- The build will compile our code, run our tests

- We can now build the new project by invoking Maven with the goals  
`clean test install site`
- The build will compile our code, run our tests, complete successful

- We can now build the new project by invoking Maven with the goals  
`clean test install site`
- The build will compile our code, run our tests, complete successful, and the same artifacts as last time are generated

- We can now build the new project by invoking Maven with the goals `clean test install site`
- The build will compile our code, run our tests, complete successful, and the same artifacts as last time are generated
- There will also be one interesting new artifact, a folder called `site`



# Building the New Project



javaExamples - Java - project\_with\_tests/pom.xml - Eclipse

Package Explore | Type Hierarchy | JUnit

project\_with\_tests [javaExamples master]

- src/main/java
  - cn.edu.hfu.iao
  - Main.java
  - package-info.java
  - src/main/resources
- src/test/java
  - cn.edu.hfu.iao
  - MainTest.java
  - src/test/resources
- JRE System Library [JavaSE-1.8]
- Maven Dependencies
- settings
- src
- target
  - apidocs
  - archive-tmp
  - generated-sources
  - generated-test-sources
  - javadoc-bundle-options
  - maven-archiver
  - maven-status
  - site
    - project-with-tests-0.0.1-jar-with-dependencies.jar
    - project-with-tests-0.0.1-javadoc.jar
    - project-with-tests-0.0.1-sources.jar
    - project-with-tests-0.0.1.jar
    - classpath
    - gitignore
    - project
    - pom.xml

project\_with\_tests/pom.xml

```
<execution>
<goals>
<goal>attached</goal>
</goals>
<phase>package</phase>
<configuration>
<descriptorRefs>
<descriptorRef>jar-with-dependencies</descriptorRef>
</descriptorRefs>
</execution>
```

Overview | Dependencies | Dependency Hierarchy | Effective POM | pom.xml

Problems | Javadoc | Declaration | Search | Console | Progress | Debug

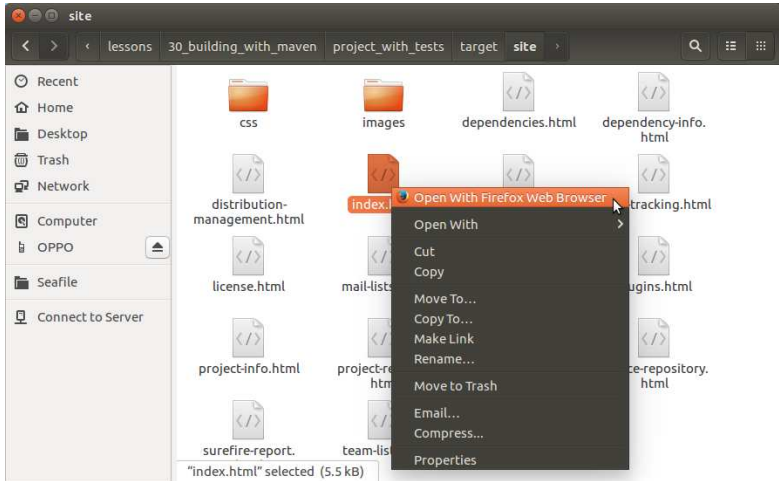
```
<terminated>project_with_tests (2) [Maven Build] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Feb 13, 2017, 2
[INFO] Skipped 'Surefire Report' report, file 'surefire-report.html' already exists for the English version
[INFO] Generating 'About' report ... maven-project-info-reports-plugin:2.7
[INFO] Generating 'Plugin Management' report ... maven-project-info-reports-plugin:2.7
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-antrun-plugin/1.3/m
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-antrun-plugin/1.3/m
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/12/maven-pl
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/12/maven-pl
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-dependency-plugin/2
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-dependency-plugin/2
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-release-plugin/2.3
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-release-plugin/2.3
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/release/maven-release/2.3.2/maven
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/release/maven-release/2.3.2/maven
[INFO] Generating 'Distribution Management' report ... maven-project-info-reports-plugin:2.7
[INFO] Generating 'Dependency Information' report ... maven-project-info-reports-plugin:2.7
[INFO] Generating 'Source Repository' report ... maven-project-info-reports-plugin:2.7
[INFO] Generating 'Mailing Lists' report ... maven-project-info-reports-plugin:2.7
[INFO] Generating 'Issue Tracking' report ... maven-project-info-reports-plugin:2.7
[INFO] Generating 'Continuous Integration' report ... maven-project-info-reports-plugin:2.7
[INFO] Generating 'Project Plugins' report ... maven-project-info-reports-plugin:2.7
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/m
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/m
[INFO] Generating 'Project License' report ... maven-project-info-reports-plugin:2.7
[INFO] Generating 'Project Team' report ... maven-project-info-reports-plugin:2.7
[WARNING] The time zone 'China Time Zone' for the developer 'Thomas Weise' is not a recognised time zone, u
[INFO] Generating 'Project Summary' report ... maven-project-info-reports-plugin:2.7
[INFO] Generating 'Dependencies' report ... maven-project-info-reports-plugin:2.7
[INFO] Generating 'Surefire Report' report ... maven-surefire-report-plugin:2.18
[WARNING] Unable to locate Test Source XRef to link to - DISABLED
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 02:19 min
[INFO] Finished at: 2017-02-13T14:56:29+08:00
[INFO] Final Memory: 47M/62MB
[INFO] -----
```

site-project\_with\_tests/target

- We can now build the new project by invoking Maven with the goals `clean test install site`
- The build will compile our code, run our tests, complete successful, and the same artifacts as last time are generated
- There will also be one interesting new artifact, a folder called `site`
- It contains a documentation of our project and the surefire test report

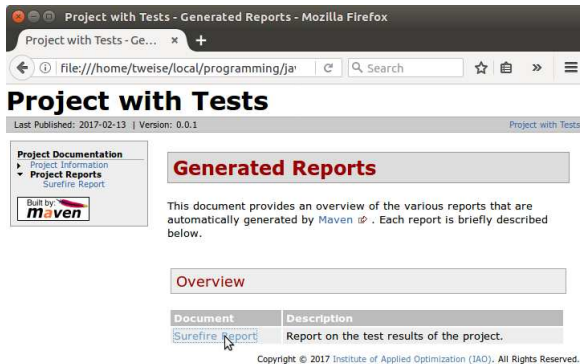
- We can now build the new project by invoking Maven with the goals `clean test install site`
- The build will compile our code, run our tests, complete successful, and the same artifacts as last time are generated
- There will also be one interesting new artifact, a folder called `site`
- It contains a documentation of our project and the surefire test report
- We click on `index.html` inside this folder

# Building the New Project



A screenshot of a Mozilla Firefox browser window. The title bar reads "Project with Tests - About - Mozilla Firefox". The address bar shows the file path "file:///home/tweise/local/programming/ja...". The page content has a header "Project with Tests" and a sub-header "About Project with Tests". A sidebar on the left lists "Project Documentation" items, with "Project Reports" highlighted. A mouse cursor points to the "Project Reports" link. Below the sidebar, a banner says "Built by: maven Project Reports". The main content area contains a paragraph: "A project similar to 'Project with Dependencies', but now also performing JUnit tests." The footer of the page states "Copyright © 2017 Institute of Applied Optimization (IAO). All Rights Reserved."

file:///home/tweise/local/programming/javaE...ject\_with\_tests/target/site/project-reports.html

A screenshot of a Mozilla Firefox browser window displaying a web page titled 'Project with Tests - Generated Reports'. The address bar shows a file path: 'file:///home/tweise/local/programming/java...'. The page has a dark header with the title and a light gray sidebar on the left. The sidebar contains a 'Project Documentation' section with links to 'Project Information' and 'Project Reports', and a 'Built by: maven' logo. The main content area has a 'Generated Reports' section with a paragraph explaining that the document provides an overview of reports generated by Maven. Below this is an 'Overview' section containing a table with two columns: 'Document' and 'Description'. The table has one row with 'Surefire Report' and 'Report on the test results of the project.' respectively. A mouse cursor is pointing at the 'Surefire Report' link. At the bottom of the page, there is a copyright notice: 'Copyright © 2017 Institute of Applied Optimization (IAO). All Rights Reserved.'

Project with Tests - Generated Reports - Mozilla Firefox

Project with Tests - Ge... x +

file:///home/tweise/local/programming/java... Search ☆ 📁 » ☰

## Project with Tests

Last Published: 2017-02-13 | Version: 0.0.1 [Project with Tests](#)

**Project Documentation**

- Project Information
- ▼ Project Reports
  - Surefire Report

Built by: 

### Generated Reports

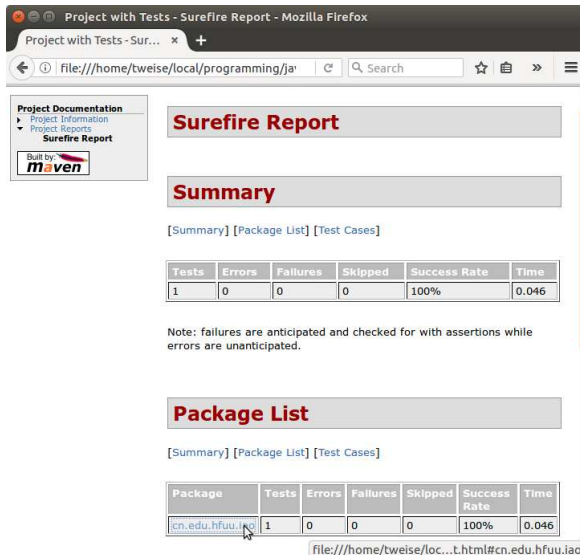
This document provides an overview of the various reports that are automatically generated by [Maven](#). Each report is briefly described below.

#### Overview

Document	Description
<a href="#">Surefire Report</a>	Report on the test results of the project.

Copyright © 2017 Institute of Applied Optimization (IAO). All Rights Reserved.

file:///home/tweise/local/programming/java/javaE...ject\_with\_tests/target/site/surefire-report.html

A screenshot of a web browser displaying a Surefire Report. The browser window has a dark title bar with the text 'Project with Tests - Surefire Report - Mozilla Firefox'. The address bar shows the file path 'file:///home/tweise/local/programming/ja...'. The page content includes a sidebar with 'Project Documentation' links, a main heading 'Surefire Report', a 'Summary' section with a table of test results, a note about failures, a 'Package List' section with another table, and a URL bar at the bottom showing a link to the report.

## Project Documentation

- Project Information
- Project Reports
  - Surefire Report



## Surefire Report

### Summary

[Summary] [Package List] [Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
1	0	0	0	100%	0.046

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

### Package List

[Summary] [Package List] [Test Cases]

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
cn.edu.hfuu.iao	1	0	0	0	100%	0.046

file:///home/tweise/loc...t.html#cn.edu.hfuu.iao

# Building the New Project



Project with Tests - Surefire Report - Mozilla Firefox

Project with Tests - Sur... x +

file:///home/tweise/local/programming/java/ Search ☆ 📁 » ☰

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
cn.edu.hfuu.iao	1	0	0	0	100%	0.046

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

cn.edu.hfuu.iao

	Class	Tests	Errors	Failures	Skipped	Success Rate	Time
	MainTest	1	0	0	0	100%	0.046

**Test Cases**

[\[Summary\]](#) [\[Package List\]](#) [\[Test Cases\]](#)

MainTest

	testFitting0011ResultsIn01	0.046
--	----------------------------	-------

file:///home/tweise/local/programming/java/java...te/surefire-report.html#cn.edu.hfuu.iaoMainTest



- We have learned about creating and building projects with Maven
- Maven is the most widely used build tool in the Java world
- Maven allows us to specify the “coordinates”, i.e., a group ID, an artifact ID, and a version, which uniquely identify one exact version of our project
- It allows us to specify other projects we depend upon by using their coordinates
- It allows us to define a complete build process, where each plugin executed is again specified by its coordinate
- It can find and download many open source libraries from the internet automatically for us (based on the coordinates in the dependency information)
- Maven also automatically resolves the dependencies of our dependencies recursively for us
- Maven builds and project configurations are thus 100% reproducible
- It can generate all kinds of artifacts for us, including Javadoc, jars, source code jars, and a HTML page describing our project
- It allows us to specify meta-data about our project, such as an URL and developer information

# 谢谢

## Thank you

Thomas Weise [汤卫思]  
tweise@hfu.edu.cn  
<http://iao.hfu.edu.cn>

Hefei University, South Campus 2  
Institute of Applied Optimization  
Shushan District, Hefei, Anhui,  
China



Caspar David Friedrich, "Der Wanderer über dem Nebelmeer", 1818  
[http://en.wikipedia.org/wiki/Wanderer\\_above\\_the\\_Sea\\_of\\_Fog](http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog)