2. Introduction to Maven

Contents

[1. Course Intro 1](#_Toc18928)

[2. Introduction 2](#_Toc6445)

[3. High Level Overview 2](#_Toc24819)

[4. Ant vs. Maven 3](#_Toc18842)

[5. Installation Best Practices 6](#_Toc28271)

[6. Demo: Hello World 7](#_Toc18310)

[7. Java 10+ 8](#_Toc31148)

[8. Demo: Java 10+ Build 8](#_Toc11884)

[9. Summary 10](#_Toc25671)

# 1. Course Intro

=>slides: Pg. 1-2

In this course, we're going to cover a brief introduction to what Maven is. This is applicable to whether you are new to Maven or you've used various other build tools such as Ant. We're going to do a detailed overview of the pros and cons between build tools such as Ant or other scripting tools and what's different about Maven. We'll cover the key concepts of Maven, concepts like convention over configuration. And don't worry if you've never heard that term before. This is the foundation for which Maven is built on, and we're going to discuss what this means. But in a nutshell. If you follow their methodology, it makes your life a lot easier. We're going to cover the day‑to‑day programming concepts and how to integrate Maven into your typical code throughout the day. We'll also cover basic integration as far as tying it into your IDE and how to make it work well with other tools that you're already using. You can use Maven as a standalone tool, or you can use it inside of an IDE such as Eclipse, Spring, or IntelliJ. We're actually going to go through a step‑by‑step instruction as to how to do that in your IDE and see how that makes development a lot easier. Maven can also be very complex. It can do a lot of things for you such as multi‑module builds and very sophisticated, tough‑structured concepts, but it can also be very simple for you to use. We're going to see if we can make this an easier tool for you to use in your day‑to‑day coding.

=>slides: Pg. 3

Topics we're going to cover, we're going to look at what general introduction to Maven is, and then we're going to dive into the structure. What folder structure does Maven like? Well, there is a very basic structure that if you follow it, everything just kind of works for you. Then we're going to dive into dependencies. If you use Maven for one thing, it should be dependency management in handling transitive dependencies inside your application. Next, we're going to dive into repositories, why we want to use repositories, what they are good for, and what your local repository is and how that would also tie into your corporate repository. Then we're going to look at plugins. Every action you do inside of Maven is a plugin. We don't need to write our own plugins, but we'll talk about how you download and use other ones than just the standard built‑in default ones and how to see what plugins are available for you out there. Then we're also going to spend some time, as I mentioned earlier, looking at the IDE integrations and how to tie that into your builds within your IDE. Let's get started by diving into the fundamentals of Maven.

# Introduction

=>slides: Pg. 4

This module is a basic introduction to Maven and what Maven is.

=>slides: Pg. 5

We're going to go through and do a high‑level overview of Maven and just look at it from a 50,000 foot view, discuss a little bit about why you'd want to use it. Then we'll dive a little deeper if you've used Ant before or you're currently doing builds out of your IDE. We'll discuss why you want to use Maven over Ant and over your IDE, what the pros and cons of each are, and we'll go into a little bit more in‑depth look of just the cons of Ant versus Maven. Next, we'll look at the installation best practices and basically how a standard installation of Maven should work, what you need to set up on your machine, and where you download it from, those types of things, and then we'll look at some sample code. We'll do a sample Hello World application that we can get the most basic thing up and running and show you how it affects your job application. And then we'll explore the POM a little bit. Not in deep detail, though, because we're going to cover that in its own separate module. Then we'll follow all of these up with a brief summary about what we've covered and what you should've learned in this module. Let's look at that high‑level overview now.

# High Level Overview

=>slides: Pg. 6

Maven at its simplest is just a build tool. If you've used tools like Ant before or just written custom batch files, it's a tool to build source code and produce an artifact or produce and have an output of it. Maven always produces one output that is called an artifact. You can think of it as a component or a jar, even a .zip file, and Maven will help us manage our dependencies. And, as mentioned earlier, that dependency management is probably the number one reason that people get into using Maven as managing all the different dependencies you need is inside an application. If you are using a tool like Hibernate, there is one Hibernate jar that you need to import, but it has 13 or more transitive dependencies. In other words, other libraries that it needs, like Commons IO, Commons Lang, those types of dependencies to be able to use that inside your application. With Maven, you can tell it that you want to use this, and it will go and download all of the transitive dependencies. We'll go into detail a little bit more in a later module on dependencies and transitive dependencies and their management. At a macro level, Maven can be used as a project management tool in the sense that it handles the management or versioning of your code. Maven has a version number associated with each component, and as I mentioned earlier, we only produce one version of an artifact. So we can use Maven to tell it that if it is version 1.0 or version 2.0, those types of clarifications on the artifact that we produce. It will also help us describe things like what our source control module is, or where our documentation is stored, who the developers are. It has some made information also tied to what your project is. The other nice benefit of using Maven for builds is that, as mentioned earlier, it can produce extra information. You can produce the Javadocs, the source code, and jar file with all of the extra information your IDE will use.

=>slides: Pg. 7

Who owns Maven? It's actually managed by the Apache Software Foundation. You can go to maven.apache.org to download it, and an interesting side note is that Maven sites are actually built with Maven. Maven's also open source, and it's freely available. You can get it there and download and change the source if you want. If you go to a Maven web page, all the layout is done with their site generation plugin. That's one of those extra benefits that you can easily produce. Like we mentioned earlier with Javadocs, you can generate a website for your code base and structure.

=>slides: Pg. 8

Why do we want to use Maven? There's a lot of reasons. Repeatable builds, so if you start talking about source control management or configuration management, or the newer term of DevOps or SysOps. It's the ability for us to be able to recreate our build for any environment, as well as to not have to change our settings for each environment. So Maven lets us externalize a lot of our settings to where we can now develop on Windows, test on Linux box, and deploy to a different flavor of Linux or Unix in production. We can also utilize tools like Docker to help us do this inside of our application. We have all of our properties externalized to where our code isn't dependent on the environment that it's being built in. Another really nice feature of Maven, and I alluded to this earlier, is transitive dependencies. Downloading a dependency will also pull other items it needs in. This is the number one reason without a doubt that people start using Maven. A third benefit is that it contains everything you need for your environment. So whether I'm building in my IDE, building from the command line, it contains all the information I need to build our code and reproduce our code. This fourth bullet point might confuse you a little bit. It works with a local repo. Historically, you've always downloaded all of your jars or other resources and kept them right inside your project, which would result into if I had 20 projects, I could possibly have the same jar downloaded at least 20 times. It needs that for each one of those environments. Well, Maven works from a local repo, just a local structure that enables me to download it once and then just reference it from there and use it from that location. It saves on disk space and saves on overhead. Not a huge deal at first, but when you start doing a lot of builds, it actually adds up really quick. Another benefit is that it works very well with your IDE. It doesn't matter if your IDE is NetBeans, IntelliJ, or Eclipse, we're going to focus on Eclipse and Spring STS in this training, but it works well with your IDE, and you can also use it standalone. What I mean by that is that I can build from the command line, and the same functions that I call from the command line are also being called by my IDE. It doesn't have a different path for the different environment. You've set up your job environment variables, and it's all derived from your Maven files. One last item is that it is the preferred choice for working with build tools like Jenkins or CruiseControl, Hudson, Bamboo, any of the various automated build tools that are out there for doing continuous integration. Maven is, without a doubt, the first class resident that they want you to use for that type of stuff. It works with all of those tools, and there is also a lot of additional plugins that you can use that are also integrated with those continuous integration tools.

# Ant vs. Maven

=>slides: Pg. 9

Let's discuss Ant versus Maven. A lot of people think that Ant and Maven are meant to compete against each other, and really, they could be used in conjunction with one another, but they're trying to solve two different problems. Ant was developed originally to be a replacement for a build tool called Make that wasn't cross platform. Make was designed to work on one platform, and we've tried to adapt it to other platforms over the years. Ant was built on top of Java and using XML, both tools that were meant to be used cross platform. So regardless of whether you're on a Windows machine, Mac, Linux, or some other environment, you can build things and have them be able to transfer from one environment to another. Make was built around a UNIX environment and was somewhat brittle, ran into problems like white space and hidden characters, it was very powerful was used for a lot of years, in fact, it's still used today a little bit. It's very brittle in nature and not very cross platform compliant. As mentioned earlier, Ant is built on top of Java and is XML based, but it's very procedural. You have a hard time inheriting anything, you have to go out of your way to use different pieces, and it's a stretch to be able to use composition or things like that inside of your Ant scripts.

=>slides: Pg. 10

With Ant, you have to explicitly code everything. You have to call out what your targets are, what goal we're going to do next if we're going to chain goals. I've got a little code snippet here what an Ant target is just for doing a clean. It's very clear that we're calling clean, we even have a description of what it is, and there is a comment thrown in there, and we're going to clearly delete this directory, that's our build directory, by the way. You can see that the build directory is a variable that we're passing in and it's going to get evaluated at runtime, but notice we have to define everything the way we want to do it. This actually can lead us to some problems. What if we like the word clean

=>slides: Pg. 11

and you like the word clear, and somebody else wants the phrase clean up for this target? It can be left to any number of variations and it's not a standard. You have to go into every file and know that this is what you need to call to perform a clean. This is what you're going to call for clear or clean up. Init is another one that we've seen people use from time to time as to what they would like to have their clean up procedure run as word delete or dissed and delete distribution and that type of stuff. So Ant leads us to a lot of variations that can be problematic.

=>slides: Pg. 12

There is also a lot of tribal or organizational knowledge that gets built into Ant, not specifically because you can rename variables or certain things, but because everybody does it a little bit different. There is not a standard out there for what things are called, each organization ends up having a large repository of these scripts that are unique to them, nothing will carry over from one job to another, or necessarily even to one project to another. If you go to share this with another project too, you have to go in and copy and paste all of these files to be able to reuse them. There is not a lot of reuse, there is not really the notion of inheritance, and there is no structure, nothing like that with Ant.

=>slides: Pg. 13

Maven is more than a scripting tool, it's a full‑featured build tool. We're going to dive more into that throughout the rest of the training, but you get a lot of implicit functionality built into Maven. Maven's clean is Maven's clean, it doesn't matter if we're deleting the target directory, generated sources, whatever we're doing, whatever we're going to clean up inside of this application, it's always going to be called the goal clean. You get a lot of consistency across projects that way. You're also able to achieve inheritance in your projects by using parent palms or an inherited palm through composition, those types of things. You also get transitive dependencies. In some of the other slides that we've covered so far, we've mentioned that if you use Maven for one reason at all, it should be for transitive dependencies, meaning that if you need to pull down a jar, it's going to pull in all the jars that it needs to work with in your application. Now you can do this using Ant and Ivy, the add‑on for Ant that tries to do dependency management, but every project that is a major project using Ivy has since migrated to a pure Maven approach. Another key point worth mentioning is that Maven is built around a versioning system. What is meant by that is that handles things, calling something at a snapshot inside of a Maven project actually has some context behind it. We've got a whole section coming up on versioning. Let's not dive into that too much right now, but it's one of the key goals that Maven had in mind when it was designed. Let's rather talk about some of the pros and cons a little bit closer and actually show you some of the things that maybe aren't so great about Maven or that it has a steeper learning curve.

=>slides: Pg. 14

Pros and cons of Maven versus Ant, Maven could be a bit of a black box, meaning that you don't necessarily see where all of the stuff is defined at, it has a little steeper learning curve and you've got the convention over configuration, meaning that if you follow Maven's convention, it works really well. But if you try to step outside of their boundaries, things really start to crumble quite a bit. There is considerably better IDE integration with Maven more so than Ant. In Ant, you call targets from the IDE and Maven has a much better, much richer integration. There is also a lot less overhead through the use of repos now. Traditionally, Ant would require us to download all of these files multiple times. With Maven, we typically use a local or corporate repository so you don't have to download the same file 20 times because you have 20 projects using it. One last pro is that it also is somewhat of a con is that Maven is a different mindset of a problem that occurs and people trying to make Maven act like Ant, one is a scripting tool and one is a build tool. Biggest complaint you'll see is, well, I can do this with Ant, but I can't do this with Maven. You need to quit making Maven try to act like Ant. Let's cover some actual pros for Ant. Ant is really clear and very straightforward, you know exactly what you're doing, and you can easily trace through your files. You can see every step through the targets that you explicitly call, and it's a lot quicker to learn, but it's also very copy and paste intensive. If you forget to change a variable or something inherited that you weren't expecting to get in an inherited file, problems occur. And finally, you have a larger project size using source control using Ant over Maven. You're going to copy those files and store them in Git Subversion or some other source code control tool like Bitbucket, and that's just the nature of what Ant does.

=>slides: Pg. 15

Let's look at a basic Ant build file now. Let's walk through an Ant build file. You can see that we've got a clean, compile, and jar target and it's a pretty straightforward build file. The clean obviously deletes our directory. Compile makes our directory first to make sure that it didn't exist and executes a javac task that compiles the code and the specified source directory into the destination directory. And lastly, we've got a jar target that gathers up everything in the build/classes directory and packages it up into a named jar file of HelloWorld.jar. Without looking at this, this looks like a complete Ant build file. There is actually a lot of problems with this file. I could call jar first because we haven't set up the previous goals that this one depends on and my compiler clean may have not ran yet. We can actually get an erroneous jar that doesn't contain the information I want to build for this application. This is actually a very brittle build file and can enable me to skip steps while running my application or building it. It's not really a good contract about what we're trying to do with our application, but as you can see, it's pretty clearly laid out of what we're intending to do. If we follow the steps, it can achieve our build. This is where we're talking about tribal knowledge that comes into play, and what we mean by that is that we have to call our Ant clean, Ant compile, and Ant jar specifically in that order for our application to work. Now that's a little bit of a false sense of security because we shouldn't have to know anything if our Build tool is doing what it's supposed to be doing, at least at that level. Let's look at a Maven POM file instead.

=>slides: Pg. 16

Now the Maven POM file can do all of those goals that we just ran in the Ant file on this same application. We can clean, compile. and package the jar and it will all work because it knows looking at our code just from this Maven POM file. This is what we're talking about that it can be kind of a black box. You don't see all the setup declaration, if you follow their conventions, their naming conventions, their directory structures, this type of information, this application will just run. So it's a little bit non‑descriptive at first until you understand the semantics of how Maven works. This is clearly a lot simpler, but it could also be more confusing at first. The convention over configuration is what we will discuss throughout the rest of this course. Let's now look at some of Maven's installation best practices.

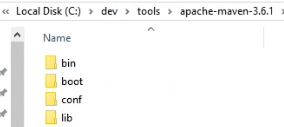
# Installation Best Practices

=>slides: Pg.

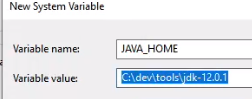
Let's look at how we install Maven and the setup and configuration of it just to get our environment up and running.

  
To get a copy of it, we just need to navigate to maven.apache.org, and you can see we have a Download link in the left‑hand section, or there's one right in the center of our page. They go to the same place. I'm going to click on that. If you ever have a problem getting Maven, you may want to check to make sure you don't have a mirror that is blocked. So some companies will block FTP sites. This is a common problem that I see while people are trying to get various applications downloaded. They'll go to an FTP site or a mirror that's blocked for whatever reason, and it will just error out, and they don't know why. So if you're having a problem downloading it, try one of the different mirrors. This one should work for us. I'm going to scroll down on the page a little bit, and you can see that we have a tar file, a binary zip archive, and the source code. A lot of times you don't really need to download the source code. In fact, I would dare say, rarely do you need to download the source code, and tarballs are usually for Linux or UNIX‑based environments, where ZIPs are more for Windows environments.

  
I'm going to go ahead and download the binary zip archive. You can see it downloading down below. I've actually already got it done on my box, so I'm just going to minimize this and let it finish. Now I want to open up File Explorer, and I like to create a folder structure of C:\dev\tools and then expand my JDK there, my IDE, Maven. Any other tools like that I have, I like to put underneath this directory. So I want to go ahead and grab that ZIP file that I've already downloaded and just copy the contents over there. Now, as it expands that, two things. You'll notice that the ZIP file actually comes over as a folder named the exact same thing inside the ZIP file. So when you click on this, it should go directly into a bin, boot, conf, lib directory. So from our tools directory, you'll see that apache‑maven. From time to time, I'll see people who have a nested Apache folder underneath there, depending on how they've expanded that. So it'll be apache‑maven‑3.6.1 and a folder inside that called apache‑maven‑3.6.1, and then it will go into that bin, conf, lib directory.

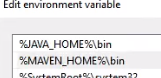
  
So just make sure your structure looks the same, that it goes into dev, tools, apache‑maven‑3.6.1, and then you've got your bin directory there. And I keep referencing the bin file because that's what we're going to now point our environment variables to. Let's go ahead and minimize these folders.

  
Let's go ahead and edit our system environment variables. Now notice I say system environment variables. If you click on the user environment variables, it's not going to let you edit those. So let's open up this account and say Environment Variables. And if these are grayed out down here, the New, Edit, and Delete buttons, it's because you've chosen to edit the user variables.



Let's go ahead and click New, and for the Variable name, we will say JAVA\_HOME. I already installed Java on this OS, but I have not set up the environment variables. And if you don't have this set up correctly, you will get an error when you try to run Maven. I'm going to click Browse Directory, and I'm going to go to This PC, and I have all of my stuff under C:, dev, tools, and I'll click my jdk for the first one and click OK. So that should say C:\dev\tools\jdk, yours may not be exactly, ‑12.0.1, do it to the directory that you installed yours to, and click OK. And then we want to do one more environment variable.

  
We'll say New, MAVEN\_HOME, and we'll Browse Directory again. We'll do the same thing. We'll go to C:, dev, tools, and we have our apache‑maven‑3.6.1 instance there. Click OK and OK again. We're almost done; we need to edit the Path.

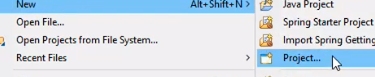


So if we come over here and say Edit Path, this brings up a different dialog. If you've used other versions of Windows in the past, it didn't look the same as this. We have to do two things here. Want to click New, and we want to type in %JAVA\_HOME%\bin. And we want to move that up to the very top. And then we want to do another New, want to do %MAVEN\_HOME%\bin, and move that up to the second position, just below JAVA\_HOME. Yours should look similar to that, %JAVA\_HOME%\bin, %MAVEN\_HOME%\bin, and click OK, click OK, and then click OK again. And now we should be able to run a Command Prompt. And from here I can type java, and I should get different commands for Java. I should be able to type mvn and get different options for Maven. The first time you run this, it's going to give you an error because it can't find a build file. That's actually correct; we don't have a project set up yet. That's what we're going to do next.

# Demo: Hello World



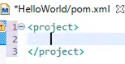
We finished off the last example showing you that if we ran Maven without a project we would get a build failure. Let's show you one more command before we dive into building our project. If I type mvn ‑version and hit Enter I'll get the printout of the Maven version that I'm using, its location, the Java version that I'm using. This is just a good, quick check for you to see exactly what your configuration is. Now let's go ahead and create that project to get our application up and running. I'm using Spring STS. You can use Spring, you can use Eclipse, you can use IntelliJ. You can honestly just use Notepad for this example if you want. I like Spring STS so I've gone ahead and used it. And when I first launch it it's going to ask me for a workspace. So I've gone ahead and created a directory of C:\dev\workspace. I want to go ahead and launch our IDE. It'll take it a second to come up. We're going to go ahead and create a project inside of here and we're going to start with just a very simple project. We're not going to choose any of the preconfigured ones for us because we want to demonstrate the simplest example we can. Okay, now that our application's up I want to go ahead



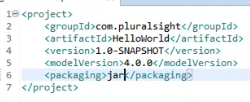
and click File, New, and just choose Project. Don't choose any of the other ones, just a simple project. Underneath there we can click Project again and click Next.

  
We're going to give it a name of HelloWorld, and click Finish. Now that this is up we can go ahead and   
right‑click on our project and choose New, File.

  
And we want to call this pom.xml. We'll select Finish. And Spring STS is going to try and convert this to a project for us and so it's going to try and download plugins on the back end and figure out what we want to do with our POM file. Now we have our empty pom.xml file. We can go ahead and start adding elements inside of here, and there's five things that we need.

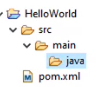


We're going to start by giving it our outermost element of project And when we close that off it will automatically put the closing tag in there for us. It's one of the nice features of using an IDE like this. We'll start with adding the groupId. And you can tell it's already identified that I'm in a XML file, and specifically I'm in a Maven POM file so it's going to start suggesting tags for me.

  
Let's start by adding our groupId. And since this is for Pluralsight we're going to do com.pluralsight. If your business was abc.org, you would put in org.abc. Next thing we'll add is our artifactId. And inside of here it's just the name of what our application's going to be, so we'll call it HelloWorld. The next thing we want to add is the version of our application that we're writing. So we're just going to type version. And we want to call this 1.0‑SNAPSHOT. And SNAPSHOT is all caps. Versions have a specific naming convention that we want to follow, and for this example we're going to do 1.0‑SNAPSHOT. The next thing we need to add is a version of our model for this POM. So we're going to do modelVersion. We want to choose 4.0.0. This has honestly always been 4.0.0 since I've ever used Maven. It does refer to this type of project that we're building. I'm not aware of anybody using anything else outside of that. Then we want to choose our packaging of type jar. So we can save this.



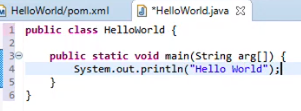
And now we can right‑click on our project and say New, Folder. Now, even though we're just doing a folder it's going to try and start treating this like a Java application. And we're going to build that out later.

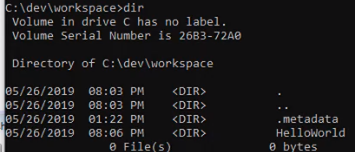


We can also add another folder underneath our src folder of main. And this is named specifically the way that it is because this is the default for Maven. And we're going to do one more folder of java. So src/main/java is the directory structure that Maven is looking for in regards to our class files.



So now we can right‑click on src/main/java and do New, File again. And we can call this HelloWorld.java. And it's going to start trying to compile things for us. So we are going to get an error that pops up as we're doing this that it wants to add it to a working set. We're just trying to get the simplest file that we can working so don't worry about that.

  
We want to call this public class HelloWorld {. And then we want to add a method in here of public static void main and pass in an array of strings, String arg. And then we're going to put a System.out.println on here, and this will give us an error when we try to add this. Out.println. It really wants to compile our application for us. We're going to just print out Hello World and then add our semicolon on the end. And that is it for what we need for our application. Let's save this file. We're going to go back out to our command line,

  
and we're going to cd to \dev\workspace. And underneath here you will see that we have our HelloWorld project.



So we're going to cd into that HelloWorld project. And now we can start off with our first Maven command. We want to run mvn clean. And I will tell you that the first time you run this it's going to go out and download a bunch of plugins and dependencies from the Maven repo. So all this stuff that's rolling off your screen, it's downloading all this stuff for the first time. So it successfully ran the clean command. And all clean does is initialize your directory structure and delete any previously generated files. So it's cleaned up our directory and set everything up for us to run it now.



The next thing we want to do is type mvn compile. And, as you might guess, this is going to compile our source code for us. So if we type mvn compile it will go out and also download more dependencies and other plugins that are out there. And you'll notice some brief pauses every now and then. That's it going out and checking against versions of current things out there, pulling new repos down, new locations of files. So Maven's going out and checking all of those dependencies and versions that we have for the configuration we currently have. Once this is done downloading all of those plugins it will then have compiled our source code for us and we can go in and run that Java application. And this only happens the first time you run it. That's the beauty of Maven, it will only download what it needs when it needs it. The unfortunate thing is the first time you run it it does download a lot of plugins for you. We ran our build, but you'll notice we got an error. This is actually an update for this course, and Maven changed a few things that it will no longer build with just the default source option. So we need to actually add a section to our POM file. Let's go talk about that right now.

# Java 10+

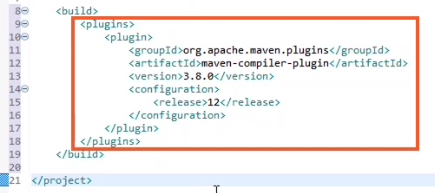
=>slides: Pg. 17

We're going to jump ahead a little bit. Since I mentioned earlier this is an update to the original course, to use Java 10 or later in your IDE, or just for general development, we have to add a build section and a plugin. If we don't add this here, you would have an error in your IDE as we just saw on the command line for all the exercises that we're going to do going forward. All we do is add a build section, followed by a plugins section, and then add the maven‑compiler‑plugin and specify a release. Let's add this to our code now.

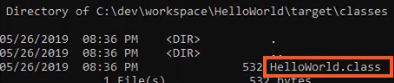
# Demo: Java 10+ Build

=>slides: Pg.

When we left off our previous example, we were getting a build failure, and we just talked about why, and that's that as of Java 10+, Java 5 was no longer supported, and the default compiled version before was Java 5. So now it's forcing you to use 7 or later because there is no longer support for 5. This is done on purpose, and it's actually helping you. It may not feel like that because you have to jump through another step to get your application to work, which previously you wouldn't have had to have done this, but they're forcing you to understand which version of code you're compiling your classes to. Let's switch back to our IDE, and inside of our POM, we want to go behind packaging, but before the closing project tag, and we're going to add a build section in here. Now, I already have this code written, and you probably don't want to watch me type it all out, so I'm going to grab it and I'm going to paste it in my application, and I'll walk through it here with you.

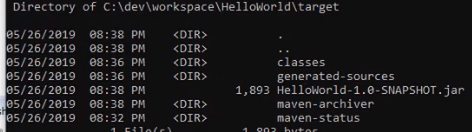
  
There's a build section, and then you create a plugin section. Now, the biggest thing I see people make a mistake doing here is they add the plugins section, but don't add the plugin element after that. So, notice that there is a nested plugins plugin section, and then inside of there we go through and add everything, just like we have before. We add a groupId, an org, apache, Maven, plugins, an artifactId of maven‑compiler‑plugin, a version of 3.8.0, and a configuration specifying a release of 12. Now I am using Java 12 in this course. If you're using Java 11, that's fine, just change your release to specify 11 versus 12 or whatever corresponding Java release you are using.

  
Let's save this and go back out to our command line and run that compilation again, and you'll see that we now have a compiled class. And we can verify that by cd‑ing down into our target directory, and you'll see a classes directory,

  
we'll cd into classes, and we'll have our HelloWorld.class here.

  
And we can, actually, because we're using the default package, just say java HelloWorld and run that, and you'll see that it prints out Hello World to our command line interface. So we have our application fully working. And let's step through those few steps that we had to do to get that working again. We created a simple project. We added a pom.xml, specified our group, artifact, version, model, and packaging type. We did have to add this build section for Java 12 in here. Then we created our src, main, java directory structure, and added our HelloWorld.java file that just simply prints out Hello World to the command line. From there we were able to run our Maven clean and compile targets that would go through and download all of the dependencies and plugins that it needed to for our project to be cleaned and compiled. We can also do the same thing and run package at that project level as well, and it will now generate our artifact for us, too.

  
So we can say mvn package, cd back into our target directory, and it will have our Hello World jar file already produced for us. If you remember, the packaging type inside of our POM was a jar. It will go ahead and create that for us.



So we can go into cd target and do a dir here, and you'll see there is our artifact idea of HelloWorld, our version of 1.0‑SNAPSHOT, our packaging type of .jar, all specified in our file.

# Summary

=>slides: Pg. 18

In this section, we covered that Ant is a very declarative, scripted build tool. We have to lay out everything, and it definitely has a shorter learning curve because we can see exactly what we're trying to do, but you do have to configure everything explicitly. You have to write out every line of code of the script to do it exactly how you want it. We covered that Maven follows a convention over configuration model, basically meaning if you're following their naming convention, things are just going to work a lot easier for you. Ant is easier to learn, but it's really only meant to be a scripting tool. Maven is centered around managing your entire project's lifecycle, so versions, how we structure our code, generate information about our application, and we're going to dive into a lot more of what these features are, but you can see the power of the model of convention over configuration versus the scripted nature of Ant from this basic example that we did. This is just a brief comparison about what the differences were between Ant and Maven. Let's dive into some of the richer features of Maven now.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*