**Maven**

Maven is a "build management tool", it is for defining how your .java files get compiled to .class, packaged into .jar (or .war or .ear) files, (pre/post)processed with tools, managing your CLASSPATH, and all others sorts of tasks that are required to build your project. It is similar to Apache Ant or Gradle or Make files in C/C++, but it attempts to be completely self-contained in it that you shouldn't need any additional tools or scripts by incorporating other common tasks like downloading & installing necessary libraries etc.

It is also designed around the "build portability" theme, so that you don't get issues as having the same code with the same build script working on one computer but not on another one (this is a known issue, we have VMs of Windows 98 machines since we couldn't get some of our Delphi applications compiling anywhere else). Because of this, it is also the best way to work on a project between people who use different IDEs since IDE-generated Ant scripts are hard to import into other IDEs, but all IDEs nowadays understand and support Maven (IntelliJ, Eclipse, and NetBeans). Even if you don't end up liking Maven, it ends up being the point of reference for all other modern builds tools.

Why you should use it

There are three things about Maven that are very nice.

Maven will (after you declare which ones you are using) download all the libraries that you use and the libraries that they use for you automatically. This is very nice, and makes dealing with lots of libraries ridiculously easy. This lets you avoid "dependency hell". It is similar to Apache Ant's Ivy.

It uses "Convention over Configuration" so that by default you don't need to define the tasks you want to do. You don't need to write a "compile", "test", "package", or "clean" step like you would have to do in Ant or a Makefile. Just put the files in the places in which Maven expects them and it should work off of the bat.

Maven also has lots of nice plug-ins that you can install that will handle many routine tasks from generating Java classes from an XSD schema using JAXB to measuring test coverage with Cobertura. Just add them to your pom.xml and they will integrate with everything else you want to do.

**Gradle**

Gradle scales

Gradle scales very well. It significantly increases your productivity, from simple single project builds up to huge enterprise multi-project builds. This is true for structuring the build. With the state-of-art incremental build function, this is also true for tackling the performance pain many large enterprise builds suffer from.

Multi-project builds

Gradle's support for multi-project build is outstanding. Project dependencies are first class citizens. We allow you to model the project relationships in a multi-project build as they really are for your problem domain. Gradle follows your layout not vice versa.

Gradle provides partial builds. If you build a single subproject Gradle takes care of building all the subprojects that subproject depends on. You can also choose to rebuild the subprojects that depend on a particular subproject. Together with incremental builds this is a big time saver for larger builds.

Many ways to manage your dependencies

Different teams prefer different ways to manage their external dependencies. Gradle provides convenient support for any strategy. From transitive dependency management with remote Maven and Ivy repositories to jars or directories on the local file system.

Step 1 − Open Eclipse Marketplace

Open the eclipse which is installed in your system. Go to help → click EclipseMarketplace.

Step 2 − Install Buildship Plugin

On the left search bar, type buildship. Buildship is a Gradle integration plugin. When you find buildship on your screen, click Install button present on the right side of the screen.

There you need to confirm the software installation by clicking the confirm button.

Step 3 − Verifying Gradle Plugin

While verifying, we will create a new project by following the given procedure. In the eclipse, go to file → click New → click Other projects.

Step 4 − Verifying Directory Structure

After successful installation of Gradle plugin, please check the demo project directory structure for the default files and folders.

SBT

Recently the revamped version 0.10 of the Simple Build Tool (sbt) was released. We are also closing in on a final 2.0.0 release of the Scala IDE for Eclipse that will hopefully bring the age of limited IDE support for Scala to an end. Having these two great tools at hand, we certainly need integration capabilities. The sbteclipse project aims at that very goal by providing a plugin for sbt that makes it possible to create Eclipse project files from an sbt project.

The current version 1.1 of sbteclipse, which requires sbt 0.10 or later, provides the additional sbt command eclipse which automatically creates Eclipse .project and .classpath files. Using the optional argument create-src will additionally create the source folders of the sbt project, e.g. src/main/scala. Of course this also works for multi-module projects. In such a case the root project often is just a folder and not a “proper” project. Therefore the optional argument skip-root can be given to prevent the Eclipse project files for the root project from being created.

As soon as the Eclipse project files are in place, the “Import...” wizard of Eclipse can be used to import “existing projects into workspace”. It is strongly recommended not to copy these projects into the workspace, i.e. simply leave the relevant checkbox unchecked. Otherwise the projects will be duplicated and possibly diverge on the long run.

sbt i.e. Simple Build Tool is a general purpose build tool written in Scala for JVM developers. It borrows good ideas from other successful build tools like Ant, Maven, and Gradle.

1. Default project layouts
2. Built-in tasks
3. Plugin architecture
4. Declarative Dependency management
5. Code over Configuration: A DSL for build tool

Apart from the feature set mentioned above sbt also provides the following additional features:

1. Interactive nature: It isn't just a build tool, it also provides an interactive environment to work in.
2. Scala REPL integration

sbt terminology consists of two terms -- task and settings. A task defines an action which you want to perform like compile. A setting is used to define a value for example name and version of the project.

With sbt whenever you want to perform any action you execute a task. Task is the unit of currency in sbt. A task can depend on another task to do its job. sbt creates a task dependency graph to determine which task should run first. If task t1 depends on task t2 then task t2 will be executed first and then task t1 will be executed. You can view all the tasks applicable to a project by running sbt tasks task.

**Comparisons**

SBT is very simple and it is focused on Scala it relies on Ivy for dependency management. Maven it's a great build tool and it enables to control the entire software lifecycle with XML files. Using the Project Object Model you can intercept all points of the software lifecycle from compile to test, packaging and deploy. Maven has it's own dependency manager. In my humble opinion the bad issue in Maven is the XML syntax, writing a POM can be annoying and too much expensive. I think we can not compare Gradle with Maven (or SBT). Gradle is built on top of Maven, Ant and Ivy. It uses Maven repositories. Gradle doesn't use XML, it's a polyglot build tool. It combines the Ant API with the Groovy language to enable developers to write a build script with an intuitive DSL. With a few lines of code you can write a Gradle build script that can do the same things that Maven can do. With Gradle you can define your own task with the Groovy language and intercept programmatically your build execution. This functional approach is not for all developers,infact Maven it's good if you don't want this behavior in your build environment. Both Maven and Gradle have plugins to integrate your build with technologies used in your projects.