# Comparison between Ant,Maven and Gradle

## Apache Ant

**In the beginning, Make was the only build automation tool**, beyond homegrown solutions**.** Make has been around since 1976 and as such, it was used for building Java applications in the early Java years.

However, a lot of conventions from C programs didn't fit in the Java ecosystem, so in time Ant was released as a better alternative.

**Apache Ant ("Another Neat Tool") is a Java library used for automating build processes for Java applications**. Additionally, Ant can be used for building non-Java applications. It was initially part of Apache Tomcat codebase and was released as a standalone project in 2000.

In many aspects, Ant is very similar to Make, and it's simple enough so anyone can start using it without any particular prerequisites. Ant build files are written in XML, and by convention, they're called *build.xml*.

Different phases of a build process are called "targets".

Here is an example of a *build.xml* file for a simple Java project with the *HelloWorld* main class:

```
1    <project>
2      <target name="clean">
3        <delete dir="classes" />
4      </target>
5
6      <target name="compile" depends="clean">
7        <mkdir dir="classes" />
8        <javac srcdir="src" destdir="classes" />
```

```
9       </target>
10
11      <target name="jar" depends="compile">
12          <mkdir dir="jar" />
13          <jar destfile="jar/HelloWorld.jar" basedir="classes">
14              <manifest>
15                  <attribute name="Main-Class"
16                      value="antExample.HelloWorld" />
17              </manifest>
18          </jar>
19      </target>
20
21      <target name="run" depends="jar">
22          <java jar="jar/HelloWorld.jar" fork="true" />
23      </target>
24  </project>
```

This build file defines four targets: *clean*, *compile*, *jar* and *run*. For example, we can compile the code by running:

```
1   ant compile
```

This will trigger target *clean* first which will delete the “classes” directory. After that, target *compile*will recreate the directory and compile src folder into it.

**The main benefit of Ant is its flexibility. Ant doesn’t impose any coding conventions or project structures.** Consequently, this means that Ant requires developers to write all the commands by themselves, which sometimes leads to huge XML build files which are hard to maintain.

Since there are no conventions, just knowing Ant does not mean we’ll quickly understand any Ant build file. It’ll likely take some time to get accustomed with an unfamiliar Ant file, which is a disadvantage compared the other, newer tools.

At first, Ant had no built-in support for dependency management. However, as dependency management became a must in the later years, Apache Ivy was developed as a sub-project of the Apache Ant project. It's integrated with Apache Ant, and it follows the same design principles.

However, the initial Ant limitations due to not having built-in support for dependency management and frustrations when working with unmanagable XML build files led to the creation of Maven.

# Apache Maven

Apache Maven is a dependency management and a build automation tool, primarily used for Java applications. **Maven continues to use XML files just like Ant but in a much more manageable way.**The name of the game here is convention over configuration.

While Ant gives the flexibility and requires everything to be written from scratch, **Maven relies on conventions and provides predefined commands (goals).**

Simply put, Maven allows us to focus on what our build should do, and gives us the framework to do it. Another positive aspect of Maven was that it provided built-in support for dependency management.

Maven's configuration file, containing build and dependency management instructions, is by convention called *pom.xml*. Additionally, Maven also prescribes strict project structure, while Ant provides flexibility there as well.

Here's an example of a *pom.xml* file for the same simple Java project with the *HelloWorld* main class from before:

```
1    <project xmlns="http://maven.apache.org/POM/4.0.0"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4          http://maven.apache.org/xsd/maven-4.0.0.xsd">
5        <modelVersion>4.0.0</modelVersion>
6        <groupId>baeldung</groupId>
7        <artifactId>mavenExample</artifactId>
```

```
8       <version>0.0.1-SNAPSHOT</version>
9       <description>Maven example</description>
10
11      <dependencies>
12          <dependency>
13              <groupId>junit</groupId>
14              <artifactId>junit</artifactId>
15              <version>4.12</version>
16              <scope>test</scope>
17          </dependency>
18      </dependencies>
19  </project>
```

However, now the project structure has been standardized as well and conforms to the Maven conventions:

```
1   +---src
2   |   +---main
3   |   |   +---java
4   |   |   |   \---com
5   |   |   |       \---baeldung
6   |   |   |           \---maven
7   |   |   |                   HelloWorld.java
8   |   |   |
9   |   |   \---resources
10  |   \---test
11  |       +---java
12  |       \---resources
```

As opposed to Ant, there is no need to define each of the phases in the build process manually. Instead, we can simply call Maven's built-in commands.

For example, we can compile the code by running:

```
mvn compile
```

At its core, as noted on official pages, **Maven can be considered a plugin execution framework, since all work is done by plugins.** Maven supports a wide range of available plugins, and each of them can be additionally configured.

One of the available plugins is Apache Maven Dependency Plugin which has a *copy-dependencies*goal that will copy our dependencies to a specified directory.

To show this plugin in action, let's include this plugin in our *pom.xml* file and configure an output directory for our dependencies:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <executions>
        <execution>
          <id>copy-dependencies</id>
          <phase>package</phase>
          <goals>
            <goal>copy-dependencies</goal>
          </goals>
          <configuration>
            <outputDirectory>target/dependencies
             </outputDirectory>
```

```
16                  </configuration>
17                </execution>
18              </executions>
19            </plugin>
20          </plugins>
21      </build>
```

This plugin will be executed in a *package* phase, so if we run:

```
1    mvn package
```

We'll execute this plugin and copy dependencies to the target/dependencies folder.

Maven became very popular since build files were now standardized and it took significantly less time to maintain build files, comparing to Ant. However, though more standardized than Ant files, Maven configuration files still tend to get big and cumbersome.

**Maven's strict conventions come with a price of being a lot less flexible than Ant.** Goal customization is very hard, so writing custom build scripts is a lot harder to do, compared with Ant.

Although Maven has made some serious improvements regarding making application's build processes easier and more standardized, it still comes with a price due to being a lot less flexible than Ant. This lead to the creation of Gradle which combines best of both worlds – Ant's flexibility and Maven's features.

# Gradle

Gradle is a dependency management and a build automation tool which **was built upon the concepts of Ant and Maven.**

One of the first things we can note about Gradle is that it's not using XML files, unlike Ant or Maven.

Over time, developers became more and more interested in having and working with a domain specific language – which, simply put, would allow

them to solve problems in a specific domain using a language tailored for that particular domain.

This was adopted by Gradle, which is using a DSL based on Groovy. **This led to smaller configuration files with less clutter since the language was specifically designed to solve specific domain problems.** Gradle's configuration file is by convention called *build.gradle.*

Here is an example of a *build.gradle* file for the same simple Java project with the *HelloWorld* main class from before:

```
1   apply plugin: 'java'
2
3   repositories {
4       mavenCentral()
5   }
6
7   jar {
8       baseName = 'gradleExample'
9       version = '0.0.1-SNAPSHOT'
10  }
11
12  dependencies {
13      compile 'junit:junit:4.12'
14  }
```

We can compile the code by running:

```
1   gradle classes
```

At its core, Gradle intentionally provides very little functionality. **Plugins add all useful features.** In our example, we were using *java* plugin which allows us to compile Java code and other valuable features.

**Gradle gave its build steps name "tasks", as opposed to Ant's "targets" or Maven's "phases".** With Maven, we used Apache Maven Dependency Plugin,

and it’s specific goal to copy dependencies to a specified directory. With Gradle, we can do the same by using tasks:

```
1  task copyDependencies(type: Copy) {
2    from configurations.compile
3    into 'dependencies'
4  }
```

We can run this task by executing:

```
1  gradle copyDependencies
```

**Ref:** https://www.baeldung.com/ant-maven-gradle