Tutorial 3: Build Management - Ant & Maven

Build Management - Ant & Maven

Part 1: Ant

Executive Summary

Most software engineers soon tire of the often repetitive command line tasks required to build a software system. Build systems such as Make and Ant automate this process and remove its tedium, while simultaneously optimizing the process to prevent unnecessary work and shorten build times. This lab will introduce the basics of using the Apache Ant build automation tool.

This tutorial is based on materials from Dr. Miryung Kim.

Installing Ant

Some systems (such as Macs) come with Ant already installed. Check to see if your system already has ant installed by attempting to run it from the command line:

ant -version

If ant is installed this command should result in Ant printing its version information. If not, you'll need to install it by **Homebrew (http://brew.sh/)** (`brew install ant`) or **MacPorts (http://www.macports.org/)** (`sudo port install apache-ant`).

Many Linux distributions have Ant available through their packaging system (e.g., `sudo apt-get install ant` should install it on Debian-based systems). Others will need to install it manually using the procedure outlined in the Ant manual available at **this link (http://ant.apache.org/manual/install.html)**.

This installation guide can sometimes be tricky, here are some abbreviated instructions for those of you using cygwin that might work:

1. Download this zip archive: **apache-ant-1.10.5-bin.zip (http://www-eu.apache.org/dist//ant/binaries/apache-ant-1.10.5-bin.zip)** or **apache-ant-1.9.13-bin.zip (http://www-eu.apache.org/dist//ant/binaries/apache-ant-1.9.13-bin.zip)**
2. Extract the archive to C:/

3. Open cygwin terminal (**installer (https://www.cygwin.com/setup-x86_64.exe)** , don't forget to install **subversion** when choosing the packages to install), do the following:

- export ANT_HOME=/cygdrive/c/apache-ant-1.9.11/
- export JAVA_HOME=/cygdrive/c/Program\ Files/Java/jdk-10.0.1
  - (or whichever jdk version you're using)
- export PATH=$ANT_HOME/bin:$PATH
- Test the ant installation by typing:

- ant -version
- The result should be similar to this:
  - Apache Ant(TM) version 1.9.11 compiled on Marc 23 2018

Ant Buildfiles

Any project wishing to use Ant must supply an XML-based buildfile (usually named "build.xml"), which specifies what Ant should do. Buildfiles consist of a top-level project element that contains at least one target sub-element and usually some number of global property sub-elements. We'll look at each of these subelements in turn before returning to how they're put together to form the top-level property element.

Global Properties

Global properties are similar to constants in that they allow you to associate a value with an identifier. These identifiers can then be used elsewhere in the buildfile to refer to the value. This is often used to define important values once that are then referenced multiple times throughout the buildfile (e.g., to define the directory where the source files are located). This concept is useful in improving buildfile readability by allowing all the important values in a buildfile to be defined in one contiguous location. It also makes it easier to modify a buildfile by enabling changes to be applied in a single location in the buildfile instead of multiple locations throughout.

Properties are defined using a "property" element. For example, the following defines the property "src.dir" to the value "src":

<property name="src.dir" value="src"/>

Once defined, a property's value can be recalled by placing its name between "${" and "}". For example, if you've defined the "src.dir" property above you can use that property elsewhere with "${my_property}". Properties can even be used in the definition of yet other properties, like so:

<property name="mypkg.dir" value="${src.dir}/com.example.mypkg"

Exercise 1:

Create the XML elements needed to define the following properties. Use properties to define other properties wherever possible.

Name	Value
src.dir	src
build.dir	build
dist.dir	dist
dist.jar.dir	dist/lib
jar.dir	lib

Targets

A target is a grouping of tasks that need to be performed to reach a certain desired state. A buildfile will typically include targets for such things as initializing the build directory structure, compiling the source files, and building a Java archive. A target is defined by a sequence of task subelements, so we'll discuss these first.

Tasks

Ant's strength lies in its powerful collection of built-in tasks. The tasks available cover pretty much anything you'll need to do during a build process, including tasks that work with archives, handle compilation, interact with version control systems, perform file/directory operations, etc. We'll cover the absolute basics for a handful of the most common tasks here, but for a comprehensive look at what is available see the **list (http://ant.apache.org/manual/tasksoverview.html)** of tasks in Ant's manual.

Tasks are defined by an element with the tasks name. The options available for each task is unique for those tasks. Usually, you need only include the bare minimum of information required to do the task and Ant takes care of the rest. Tasks also are typically aware of whether or not they need to be run again. For example, the compilation tasks won't actually compile unless the class files are out of date. Here are some of the most commonly used tasks:

mkdir

The mkdir task is used to create a directory. You need only supply the name of the directory you wish to create, for example:

<mkdir dir="${some.dir}"/>

This will create a new directory named whatever the value of the "some.dir" property is.

Exercise 2:

Write the task element necessary to create a directory named "build".

delete

The delete task is used for deleting files and directories. You supply the name of the directory or file to delete. Here are a couple of examples:

<delete dir="some_dir"/>

<delete file="some_file"/>

The first task will delete the directory "some_dir" and all of its contents; the second will delete the file "some_file".

Exercise 3:

Write the task element necessary to delete a directory named "build".

javac

The javac task is arguably the workhorse of the Ant build system; it compiles a Java source tree. You must provide a directory containing the source, and usually a destination for the compiled class files is also specified. You may also need to specify a classpath for compilation if the source to be compiled relies upon classes defined elsewhere. Lastly, you also will usually want to set the attribute "includeantruntime" to "false" which will ensure that only the classpath specified in the antfile is used and not any system or ant-specific classpath; this makes it more likely that your buildfile will work on all systems, not just yours. An example javac task follows:

<javac srcdir="src" destdir="build" classpath="my_lib.jar" includeantruntime="false"/>

This task will compile all the Java files in the "src" directory and place the compiled class files in the "build" directory whilst mirroring the internal directory structure of the "src" directory (e.g., if there is a java file at "src/com/example/MyClass.java" the generated class file will be placed in "build/com/example/MyClass.class". The compilation will use a classpath of "my_lib.jar", so any classes found in that JAR can be used by the source without problems, and the classpath will not be polluted by any other classpath entries. Not bad for a one-liner!

Exercise 4:

Write the task element necessary to compile all the java source files in the "src" directory, and place the resultant class files in the "build" directory. Be sure to use any relevant properties you previously defined. The element should also include the jarfile "my_jarfile.jar" in the compilation task's classpath.

jar

The jar task handles the process of packaging up a set of class files into a Java archive (JAR) file. You need only specify the output jarfile's name and the base directory of the class hierarchy to be packaged up. Here's an example:

<jar jarfile="my_lib.jar" basedir="build"/>

This packages up all the class files in the "build" directory and creates a valid Java archive named "my_lib.jar".

Exercise 5:

Create the task element necessary to create the jarfile named "my_jar.jar".

java

The java task allows you to execute a Java program. You must specify either a classname or an executable jarfile. If you specify a jarfile, you must also set the "fork" attribute to true (which means that a separate Java instance than the one Ant itself is running in will be created. Here's an example:

```
<java jar="executable_jar.jar" fork="true"/>
```

This executes the "executable_jar.jar" jarfile in its own Java virtual machine.

Exercise 6:

Create the task element necessary to execute the jarfile named "my_jar.jar".

We'll return now to the topic of targets. All buildfiles must contain at least one target. Targets are created using a target element which consists of some number of tasks that will be executed in sequential order whenever the target is to be built. Here's an example target:

```
<target name="clean">
  <delete dir="build_dir">
  <delete file="temp_file">
</target>
```

Now whenever the "clean" target is run, the directory "build_dir" and the file "temp_file" will be deleted. Notice that the target has a "name" attribute; every target must have a unique name. You can also specify dependencies between targets. This tells ant that before a target's tasks can be run that the target that it depends upon must have been executed. Ant keeps track of these dependencies and ensures that they are met during the build process. For example, say you have the following snippet in your buildfile:

```
<target name="jar", depends="compile">
  ...
</target>
```

Before performing the tasks contained in the "jar" target (not shown in the snippet above), Ant will make sure to run the "compile" target. This is very useful because you can request that Ant build the "jar" target and it will automatically make sure that it is being built from up-to-date class files.

Projects

Now that we know the basic components of a buildfile, we can create the required top-level project element. The project element can include the following attributes:

- name: this provides a name for the project
- default: this specifies the default target to be built when Ant is invoked without a requested target
- basedir: this specifies a base directory upon which relative paths specified elsewhere in the buildfile will be based

Here's an example project element (lower level elements are not included for simplicity):

```
<project name="MyProject" basedir="." default="jar">
  <property name="src.dir" value="src"/>
  <target name="clean">
    ...
  </target>
  <target name="compile">
    ...
  </target>
  <target name="jar" depends="compile">
    ...
  </target>
  <target name="run" depends="jar">
    ...
  </target>
</project>
```

This defines a project with one global property ("src.dir") and four targets ("clean", "compile", "jar", and "run"). The default target in this project is "jar", and you can also see how targets can be set up to depend upon one another.

Running Ant

Ant is invoked by typing "ant" at the command line. By default Ant will search for a default target (set via the top-level project element) if you don't specify a target argument. Ant also supports requesting that a specific target be built by invoking ant with the requested target's name. For example:

ant run

This would cause the run target to be built. In the example project from above, this would cause the following build sequence (due to dependencies): "compile" -> "jar" -> "run".

Putting it all together

Using the elements introduced above, you'll now use Ant to build yet another CircleCalc program.

Exercise 7:

This exercise should be done *entirely* outside of any Java IDE, e.g. Eclipse or IntelliJ IDEA.

Clone the Ant-based CircleCalc project from the following GitHub URL:

https://github.com/ut-ee461l/tutorial-ant (https://github.com/ut-ee461l/tutorial-ant)

This directory contains an incomplete Ant buildfile. Your task is to fix it. The final buildfile should meet the following requirements:

- All directory and file names referenced from within tasks shall use properties for their values
- All properties should be defined in the file before any targets are defined
- The buildfile shall include the following targets:
  - init: creates the build directory used for storing class files
  - compile: compiles all the source files in the source directory and places them in the build directory
  - dist: creates a directory for distributing the program and creates a Java archive (jarfile) that is placed in the distribution directory
  - run: executes the executable-jar created by the dist target
  - clean: removes all files and directories created by the build process
- The default target shall be dist
- All targets shall have the appropriate dependencies necessary for them to complete successfully
- Invoking "ant run" should successfully execute the CircleCalc program

Important Notes:

- The "dist" target found in the incomplete buildfile shows the additional subelements required for the "jar" task to create an executable jarfile which executes the "Circle" class
- This version of CircleCalc depends upon a class defined by the Apache Common math library. The required jar is intentionally not included in the subversion repository download. You can find the required jarfile "commons-math-2.2.jar" **(be sure to get the version 2.2!)** in the binary archives available at **this link** (http://commons.apache.org/proper/commons-math/download_math.cgi) .

And One More, For Fun...

Exercise 8:

In Eclipse or IntelliJ IDEA, create a new Java project called HelloWorld. Within this project, create a new package called usa.texas.austin and, within that package a new class called HelloWorld.java. Make your java file functional (e.g., mine just prints a message).

For your new project and within your IDE, create a new Ant buildfile (you can call it whatever you want, but build.xml is the standard choice). On:

- **Eclipse**: Open the buildfile with Eclipse's Ant Editor. Eclipse has autocomplete feature
- **IntelliJ IDEA**: Enable Ant Build Tool by View > Tool Windows > Ant Build. Add your buildfile to the window and from there you can run each target by double-clicking. IntelliJ also has autocomplete feature.

Create at least three meaningful (and useful) targets within your Ant buildfile. Convince yourself that it works.

Demo your resulting buildfile and its use to the TA. You should (1) show the TA your build file within the IDE outline view (you should find this; it's cool and handy). From looking at the outline view, the TA may ask you to invoke any one or more of your ant targets. You should be able to invoke the requested target (from within IDE) and then demonstrate to the TA (perhaps by looking at the filesystem outside of IDE) what happened when the command was invoked.

If you would like help with these steps, there are lots of tutorials; I recommend going straight to the Eclipse documentation (help.eclipse.org) or IntelliJ IDEA documentation (www.jetbrains.com/help/idea/ant.html)

What to Submit:

For this section, you should submit the XML build file that results from Exercise 7. You should also demonstrate Exercise 8 for the TA as described above.

Part 2: Maven

Executive Summary

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting, and documentation from a central piece of information. Maven offers:

- **simple project setup that follows best practices:** Maven tries to avoid as much configuration as possible, by supplying project templates (named *archetypes*)
- **dependency management:** it includes automatic updating, downloading and validating the compatibility, as well as reporting the dependency closures (known also as transitive dependencies)
- **isolation between project dependencies and plugins:** with Maven, project dependencies are retrieved from the *dependency repositories* while any plugin's dependencies are retrieved from the *plugin repositories,* resulting in fewer conflicts when plugins start to download additional dependencies
- **central repository system:** project dependencies can be loaded from the local file system or public repositories, such as **Maven Central** **(https://search.maven.org/classic/)**

The information in this tutorial is based on:

- **https://maven.apache.org/** **(https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html)**
- **https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html** **(https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html)**
- **https://www.baeldung.com/maven** **(https://www.baeldung.com/maven)**
- **http://tutorials.jenkov.com/maven/maven-tutorial.html** **(http://tutorials.jenkov.com/maven/maven-tutorial.html)**

This tutorial can be done with terminal or cmd prompt only, without an IDE.

Installing Maven

To install Maven into your computer, you can refer to this **Maven download page** **(http://maven.apache.org/download.cgi)** and follow the instructions there.

Windows

1. Ensure **JAVA_HOME** environment variable to point the Java JDK in your system
2. Download **apache-maven-3.6.0-bin.zip** **(https://www-eu.apache.org/dist/maven/maven-3/3.6.0/binaries/apache-maven-3.6.0-bin.zip)** and extract
3. Add the **bin** directory of the created directory **apache-maven-3.6.0** to the **PATH** environment variable
4. Check the installation: "mvn **-v**" in cmd prompt

UNIX

1. Ensure **JAVA_HOME** environment variable to point the Java JDK in your system
   If not, you can use this command to set JAVA_HOME:
   *export JAVA_HOME="/*usr/*lib/*jvm*/java-1.8.0*-openjdk-*amd64"*

2. Download [apache-maven-3.6.0-bin.tar.gz](https://www-eu.apache.org/dist/maven/maven-3/3.6.0/binaries/apache-maven-3.6.0-bin.tar.gz) (https://www-eu.apache.org/dist/maven/maven-3/3.6.0/binaries/apache-maven-3.6.0-bin.tar.gz) and extract
3. Add the **bin** directory of the created directory **apache-maven-3.6.0** to the **PATH** environment variable
   If not, here is the command:
   export PATH=/opt/apache-maven-3.6.0/bin:$PATH
4. Check the installation: "mvn **-v**" in terminal

MacOS (with Brew)

1. Run: "**brew install maven**"
2. Check the installation: mvn -v in terminal

Create a Project

To start a project, we could use a maven command to generate a project with [standard project structure](https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html) (https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html). Here is the command:

```
$ mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=app-name -DarchetypeArtifactId=maven-a
rchetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false
```

If you have just installed Maven, it may take a while on the first run. This is because Maven is downloading the most recent artifacts (plugin jars and other files) into your local repository. You may also need to execute the command a couple of times before it succeeds. This is because the remote server may time out before your downloads are complete. Don't worry, there are ways to fix that.

```
$ cd app-name
```

Under this directory, the project structure will look like something like this:

```
app-name
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
```

```
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

This generated project only has a simple Hello World app with a test directory.

POM File

```
$ cat pom.xml
```

The POM file in the project root directory is the main configuration file for a Maven project. It contains the majority of information required to build a project. Typical POM files have these components:

1. **Project Identifier**
   Includes: groupId, artifactId, version, packaging

2. **Dependencies**
   These external libraries that a project uses are called dependencies. The dependency management feature in Maven ensures automatic download of those libraries from a central repository, so you don't have to manage them, typically in the form of jar files, locally.

   ```
   <dependency>
       <groupId>org.springframework</groupId>
       <artifactId>spring-core</artifactId>
       <version>4.3.5.RELEASE</version>
   </dependency>
   ```

   Notice that in POM file, jUnit is added as a dependency by default.

3. **Repositories**
   A repository in Maven is used to hold build artifacts and dependencies of varying types. The default local repository is located in the .m2/repository folder under the home directory of the user. The default central repository is **Maven Central** (https://search.maven.org/).

   ```
   <repositories>
       <repository>
           <id>JBoss repository</id>
           <url>http://repository.jboss.org/nexus/content/groups/public/</url>
       </repository>
   </repositories>
   ```

4. **Properties**
   Custom properties can help to make your pom.xml file easier to read and maintain. In the classic use case, you would use custom properties to define versions for your project's dependencies. Maven properties are value-placeholders and are accessible anywhere within a pom.xml by using the notation ${name}, where name is the property.

```
<properties>
    <spring.version>4.3.5.RELEASE</spring.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
</dependencies>
```

5. **Build**
   The build section is also a very important section of the Maven POM. It provides information about the default Maven goal, the directory for the compiled project, and the final name of the application. The default output folder for compiled artifacts is named target, and the final name of the packaged artifact consists of the artifactId and version, but you can change it at any time.

6. **Reporting**
   Maven has several reports that you can add to your web site to display the current state of the project. These reports take the form of plugin goals, just like those used to build the project. Maven Project Info Reports Plugin provides many standard reports by extracting information from the POM, for example Continous Integration, Dependencies, etc. To find out more please see to the **Project Info Reports Plugin (https://maven.apache.org/plugins/maven-project-info-reports-plugin/)**.
   To add these reports to your site, you must add the plugin to the <reporting> element in the POM. The following example shows how to configure the standard Project Info Reports that display information from the POM in a friendly format:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <version>2.6</version>
    </plugin>
  </plugins>
</reporting>
```

Maven Phases / Target

These are the mostly used Maven phases:

- *validate* – checks the correctness of the project
- *compile* – compiles the provided source code into binary artifacts
- *test* – executes unit tests
- *package* – packages compiled code into an archive file, e.g. jar
- *integration-test* – executes additional tests, which require the packaging
- *verify* – checks if the package is valid
- *install* – installs the package file into the local Maven repository
- *deploy* – deploys the package file to a remote server or repository

With two additional phases:

- *clean* **–** cleans up artifacts created by prior builds
- *site* **–** generates site documentation for this project

Exercise 9

Log the output for this exercise and the subsequent exercises. You can use the "script" command or manual copy-paste. Please try **all the phases above (except deploy)** and ensure that they can run successfully. Example:

```
$ mvn compile
```

Run Application

The application can be run by at least these two different methods:

1. Generate a jar file and run the jar file manually

```
$ mvn package
$ java -jar target/app-name-1.0-SNAPSHOT.jar
```

To use this method, you have to specify the main manifest attribute in the POM file first.

```
...
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.0.2</version>
  <configuration>
    <archive>
      <manifest>
        <mainClass>com.mycompany.app.App</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
...
```

2. Compile and run with **exec-maven-plugin**

```
$ mvn compile
$ mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```

If you use this often, you can add the main class in the dependency.

```
...
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.2.1</version>
  <configuration>
    <mainClass>com.mycompany.app.App</mainClass>
    <arguments>
      <argument>foo</argument>
      <argument>bar</argument>
    </arguments>
  </configuration>
</plugin>
..
```

and just run:

```
$ mvn compile
$ mvn exec:java
```

Exercise 10

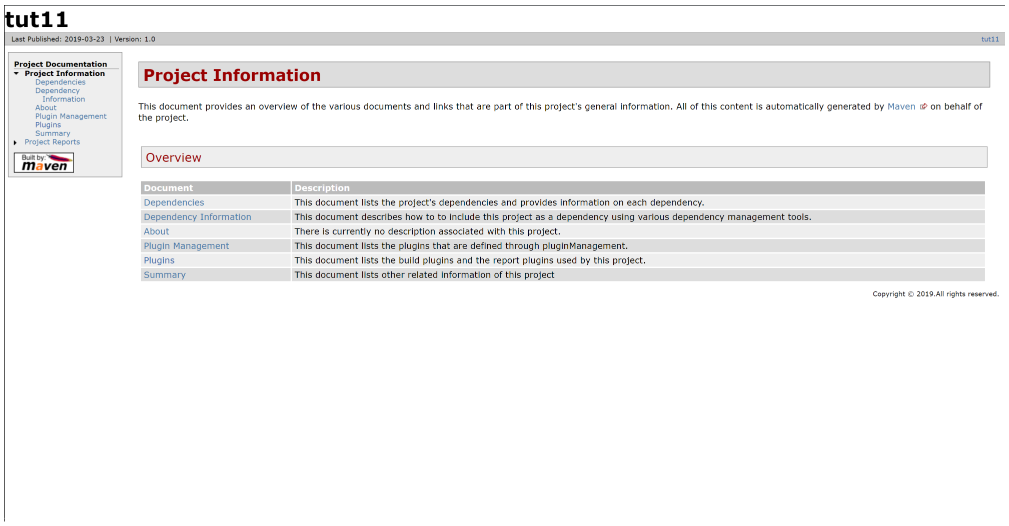
Run your application with **the two methods above** and log down the results.

Exercise 11

Download **Circle.java** **(https://github.com/ut-ee461l/tutorial-ant/blob/master/CircleCalcAntStub/src/Circle.java)** and **MyMath.java** **(https://github.com/ut-ee461l/tutorial-ant/blob/master/CircleCalcAntStub/src/MyMath.java)** . Build a Maven project with both files as the source codes.
In this exercise, you will have to:

- Set the project identifiers:
  - groupId = org.ee461l.tutorial
  - artifactId = tutmaven
  - version = 1.0
- Add the dependency used by the application. Hint: commons-math-2.2.jar
- Add the package name in the source files.
- Edit the POM file so that:
  - "mvn **validate**", "mvn **compile**", "mvn **test**", "mvn **verify**" run successful
  - "mvn **package**" generates a runnable jar file, with main manifest attribute defined. Do check that you can run the jar file.
  - "mvn **site"** generates both maven and java documentation (**maven-javadoc-plugin).** The java documentation should **only report the public attributes** of the classes, NOT the private attributes.

Maven Doc



tut11

Last Published: 2019-03-23 | Version: 1.0 tut11

Project Documentation
- Project Information
  - Dependencies
  - Dependency Information
  - About
  - Plugin Management
  - Plugins
  - Summary
- Project Reports

Project Information

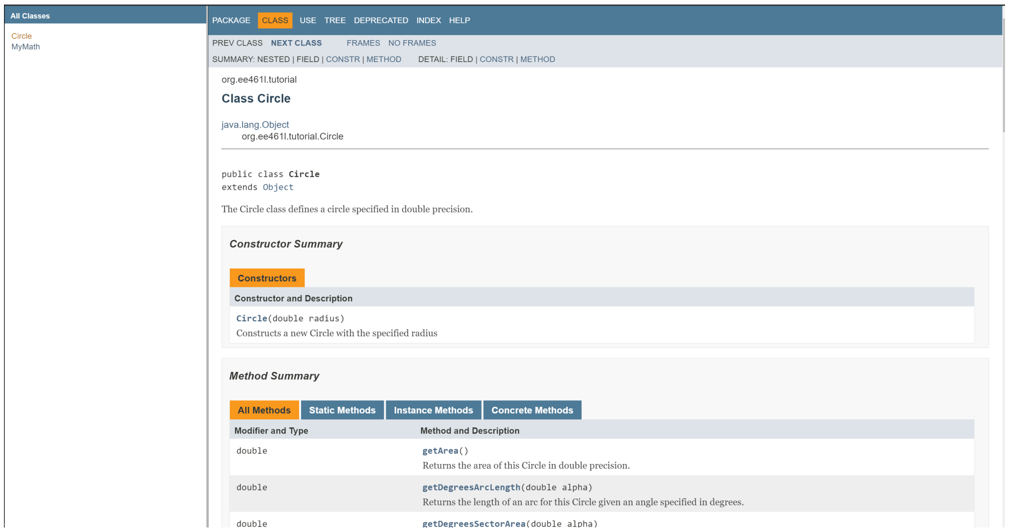
This document provides an overview of the various documents and links that are part of this project's general information. All of this content is automatically generated by Maven on behalf of the project.

Overview

Document	Description
Dependencies	This document lists the project's dependencies and provides information on each dependency.
Dependency Information	This document describes how to to include this project as a dependency using various dependency management tools.
About	There is currently no description associated with this project.
Plugin Management	This document lists the plugins that are defined through pluginManagement.
Plugins	This document lists the build plugins and the report plugins used by this project.
Summary	This document lists other related information of this project

Copyright © 2019.All rights reserved.

Java Doc



All Classes

Circle
MyMath

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.ee461l.tutorial

Class Circle

java.lang.Object
    org.ee461l.tutorial.Circle

```
public class Circle
extends Object
```

The Circle class defines a circle specified in double precision.

Constructor Summary

Constructors

Constructor and Description
`Circle(double radius)` Constructs a new Circle with the specified radius

Method Summary

All Methods Static Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
`double`	`getArea()` Returns the area of this Circle in double precision.
`double`	`getDegreesArcLength(double alpha)` Returns the length of an arc for this Circle given an angle specified in degrees.
`double`	`getDegreesSectorArea(double alpha)`

What to Submit, part the second:

For this section, you should make a zip file containing: the log for Exercise 9-11 and the project directory of Exercise 11. Name the zip file FIRSTNAME_LASTNAME.zip.