Show Submission Credentials

P0. Intro to Maven and Checkstyle

Maven and Checkstyle Primer

- ✔ Apache Maven
- ✔ Introduction to Lombok
- ✔ Introduction to Apache Commons
- ✔ Checkstyle

Apache Maven

Introduction to Apache Maven

Apache Maven (https://maven.apache.org/) is a tool used to build and manage Java based projects. The stated objectives of Maven are to:

> - Making the build process easy
> - Providing a uniform build system
> - Providing quality project information
> - Providing guidelines for best practices development
> - Allowing transparent migration to new features

You can find an expanded explanation of these objectives in the What is Maven (https://maven.apache.org/what-is-maven.html) page. The most immediate benefit of using Maven comes from the ability to easily import external Java dependencies and to explicitly determine the version of your dependencies.

Maven maintains a project object model (POM) file, which specifies the project's dependencies, plugins to run, project artifacts (i.e. JARs, WARs), and other project level attributes. An example POM file will be discussed below.

Installing Maven

Refer to the Maven download site (https://maven.apache.org/download.cgi) and installation steps (https://maven.apache.org/install.html) for platform specific instructions on installing Maven. For Linux systems, the following steps will install Maven:

> Information
>
> Maven can be installed using `apt-get` by running `sudo apt-get install maven`

1. Download the latest version of the Maven binary (bin.tar.gz) from the download site (https://maven.apache.org/download.cgi).
2. Set the `JAVA_HOME` environment variable to be the directory which contains the `bin` folder of your JDK.

   ```
   export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/Contents/Home
   ```

3. Untar the Maven binary.

   ```
   tar xzvf apache-maven-3.6.3-bin.tar.gz
   ```

4. Add Maven to the `PATH` environment variable. Replace `/opt` with the directory where you installed Maven.

   ```
   export PATH=$PATH:/opt/apache-maven-3.6.3/bin
   ```

5. Confirm the installation by checking the Maven version.

   ```
   mvn --version
   ```

Creating Your First Maven Project

The Maven documentation provides an introductory application (https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html). The following section will step through creating the quick start application and explain each step.

1. Confirm that Maven is correctly installed and accessible on the `PATH` by checking Maven's version

   ```
   $ mvn --version
   Apache Maven 3.6.3 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T11:41:47-05:00)
   ...
   ```

2. Use the Maven command line to create the quick start application. When the build succeeds, there will be a new directory name `my-app` containing the application source and a `pom.xml` file.

```
$ mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -
DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -Di
nteractiveMode=false
[INFO] Scanning for projects...
Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugin
s/maven-install-plugin/2.4/maven-install-plugin-2.4.pom
Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/
maven-install-plugin/2.4/maven-install-plugin-2.4.pom (7 KB at 15.5 KB/se
c)
...
[INFO] BUILD SUCCESS

$ ls my-app/
pom.xml src
```

3. Change to the `my-app` directory and explore the directory structure.

```
$ cd my-app/
```

4. Build the project by executing the `package` phase. Other phases include compile, clean, and test. This command must be run in the same directory as the `pom.xml` file. For this project, the `package` phase will compile the code, run unit tests, and build a JAR file.

```
$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------
-----
[INFO] Building my-app 1.0-SNAPSHOT
...
[INFO] Building jar: .../my-app/target/my-app-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------
-----
[INFO] BUILD SUCCESS
```

5. Use the `java` command to run the `App` class in the recently build JAR. You will find the `Hello World!` message printed to the console.

```
$ java -cp target/my-app-1.0-SNAPSHOT.jar com.mycompany.app.App
Hello World!
```

The POM File

In this section, we will examine the `pom.xml` file used to define the quick start project from the previous section. The POM file is placed in the root directory of the project and `mvn` command should be run in this directory. The `pom.xml` file for the quick start project is reproduced below.

```
<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>com.mycompany.app</groupId>
 <artifactId>my-app</artifactId>
 <packaging>jar</packaging>
 <version>1.0-SNAPSHOT</version>

 <name>my-app</name>
 <url>http://maven.apache.org</url>

 <dependencies>
  <dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>3.8.1</version>
     <scope>test</scope>
   </dependency>
 </dependencies>
</project>
```

The first section of the project defines a **groupId**, **artifactId**, **packaging**,and **version**. Refer to the naming conventions documentation (https://maven.apache.org/guides/mini/guide-naming-conventions.html) for recommendations on defining these fields.

- **groupId** - Used to uniquely identify this project across all projects; generally the common Java package from your project (i.e. `com.example.demos`).
- **artifactId** - Defines the jar name without a version.
- **packaging** - The type of artifact to be generated by Maven; for example `jar` or `war` .
- **version** - The version of the package being built; commonly `1.0-SNAPSHOT` , `1.0` , `1.1` , `2.0` .

The next section defines the `dependencies` this project depends on. The `dependencies` list contains individual `dependency` definitions. Each dependency is referred to by its **groupId**, **artifactId**, and **version**. For a more in depth discussion of Maven's dependency management features, consult the Introduction to the Dependency Mechanism (https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html) documentation.

Build Lifecycle and Phases

The Introduction to the Build Lifecycle (https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html) documentation provides an overview of the build lifecycle concept. Generally speaking, the lifecycle provides a definition for building and deploying an artifact. The built in lifecycles are `default` (for deploying a project), `clean` (for cleaning a project), and `site` (for creating the project's site documentation).

Each build lifecycle is composed of build phases, which represents a stage in the lifecycle. Examples of common phases include:

- `compile` - Compile the project's source code
- `test` - Test the compiled source code with a unit testing framework.
- `package` - Packages the compiled code in a distributable format, such as a JAR.
- `install` - Installs the package into the local repository.

A common workflow for developers is to `clean` and `package` their projects into a JAR. This can be done by running the following command:

```
$ mvn clean package
```

After a successful build, there should be a `target/` directory containing the packaged artifacts for the project. Refer to the Maven documentation (https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html) for more details on the build lifecycles, plugins, and project packaging.

IDE Integration

Maven has integration with a number of Java text editors including `Eclipse IDE`, `JetBrains IntelliJ IDEA`, and `Netbeans IDE`. Use of these plugins with your IDE will allow for simpler project creation, searching of dependencies, and building projects.

It is recommended to become familiar with how Maven works within your IDE. For instructions on using these plugins, consult the IDE Integration (https://maven.apache.org/ide.html) page.

Show Me the Code

If you prefer learning from examples. We offer you a POM template which enables "uber-jar" as a quick start.

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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <properties>
    <!-- always set file encoding explicitly to make the build system-independen
t -->
    <project.build.sourceEncoding>UTF-8&lt;/project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8&lt;/project.reporting.outputEncoding
>
  </properties>
  <!-- all the dependencies should be added within the dependencies tag-->
  <dependencies>
    <!--
     |
     |  TODO: Please search and add dependencies you think necessary
     |
     -->
  </dependencies>
  <build>
    <!-- the relative path from pom.xml to the source directory -->
    <sourceDirectory>src/main/java</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.6.1</version>
        <configuration>
          <!-- enable Java 8 language features -->
          <source>1.8</source>
          <!-- make compiled classes to be compatible with JVM 1.8-->
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <!-- package the artifact in an uber-jar. -->
        <!-- an uber-jar contains both your package and all its dependencies in
one single JAR file -->
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.0.0</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <minimizeJar>false</minimizeJar>
              <!-- TODO: set the filename of your jar-->
              <!-- the jar will locate at target/your_jar.jar -->
```

```
            <finalName>your_jar</finalName>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
 </build>
</project>
```

A corresponding project file hierarchy is as follows:

```
project_root
|-- pom.xml
`-- src
    `-- main
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        |-- App.java
                        `-- YetAnotherApp.java
```

Compared to the hierarchy generated by `mvn archetype:generate`, there is no test root folder `src/test/java` or the unit test code `src/test/java/AppTest.java` which requires the dependency on `junit`. If you want to adapt the `pom.xml` to the project generated by `mvn archetype:generate`, remember to add the dependency on `junit`.

Search and add dependencies

You can add the dependency of Google Gson library by the following steps:

1. Visit Maven Repository (https://mvnrepository.com/)
2. Type "gson" in the search box which has the placeholder "Search for groups, artifacts or categories"
3. Click the first result and navigate to `com.google.code.gson » gson`
4. Click the version you will use, e.g. `2.8.1`

The dependency tag is available for you, e.g.:

```
<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.1</version>
</dependency>
```

Paste the tag to your `pom.xml` and you can use Google Gson library now. If you are using IDEs with powerful Maven integration, many features will get aware of the new library, such as auto-completion.

Compile, package and execute your JAR

If you get both the source code and `pom.xml` ready, you can use the following commands to execute your program:

```
# change the shell working directory to project_root first
$ mvn clean package
$ java -cp target/your_jar.jar com.mycompany.app.App
$ java -cp target/your_jar.jar com.mycompany.app.YetAnotherApp
```

Note:

You should use the uber-jar `your_jar.jar` which packages all the dependencies to the JAR file, instead of `my-app-1.0-SNAPSHOT.jar` .

If you get a `Could not find or load main class` error, we expect you to explore solutions for such errors. If you create a post on Piazza about `Could not find or load main class` errors, we expect you to show what you have explored first, otherwise our answer will be a set of questions rather than answers. You can check the following first:

1. Do not forget to include the package in the `java -cp` command, e.g. "com.mycompany.app" in the example.
2. In the source code of `App.java` , there is the package name as `package com.mycompany.app;` .
3. The relative path of `App.java` to the source root folder (`src/main/java`) is `com/mycompany/app/App.java` .

Be careful about relative paths of File I/O

Suppose you want to open a file in `App.java` using the following code:

```
try (BufferedReader br = new BufferedReader(
        new InputStreamReader(new FileInputStream('input.txt'), StandardCharset
s.UTF_8))) {
        // process the input
}
```

with the command:

```
$ java -cp target/your_jar.jar com.mycompany.app.App
```

The following project hierarchy will show which is the correct position of `input.txt` :

```
project_root # current working directory
|-- pom.xml
|-- src
|   `-- main
|       `-- java
|           `-- com
|               `-- mycompany
|                   `-- app
|                       |-- App.java
|                       |-- YetAnotherApp.java
|                       `-- <wrong position>
|-- target
|   |-- your_jar.jar
|   `-- <wrong position>
`-- <correct postion>
```

However, if you change the working directory to `target` folder with `cd target`, and run your program with `java -cp your_jar.jar com.mycompany.app.App`. The position that `input.txt` refers to has changed:

```
project_root
|-- pom.xml
|-- src
|   `-- main
|       `-- java
|           `-- com
|               `-- mycompany
|                   `-- app
|                       |-- App.java
|                       |-- YetAnotherApp.java
|                       `-- <wrong position>
|-- target # current working directory
|   |-- your_jar.jar
|   `-- <correct position>
`-- <wrong postion>
```

To sum up, relative paths are relative to the current working directory (`pwd`), not relative to the location of the jar.

Failing to solve the correct relative paths will cause `java.io.FileNotFoundException: filename (No such file or directory)`. If so, **DO NOT** create posts on Piazza and try to fix the error by yourself.

Specify a main class of the generated JAR

There are many ways to specify a main class of the jar, so that you can execute the jar with `java -jar your_jar.jar`. One of the approaches is to configure the main class with `maven-shade-plugin`, the plugin we are using to package the jar.

```
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>3.0.0</version>
    <executions>
      <execution>
        ...
        <configuration>
          <minimizeJar>false</minimizeJar>
            <!-- specify a main class with transformer(s) tags -->
            <transformers>
              <transformer implementation="org.apache.maven.plugins.shade.resour
ce.ManifestResourceTransformer">
                <!-- do not forget the package name -->
                <mainClass>com.mycompany.app.App</mainClass>
              </transformer>
            </transformers>
          <finalName>your_jar</finalName>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

References

- Apache Maven Home Page (https://maven.apache.org/)
- Maven Download (https://maven.apache.org/download.cgi)
- Maven Install (https://maven.apache.org/install.html)
- Maven Technical FAQ (https://maven.apache.org/general.html)
- Maven Documentation Index (https://maven.apache.org/guides/index.html)

Introduction to Lombok

Introduction to Lombok

We always talk about object-oriented programming. It has great advantages over other programming styles which helps you to design reusable, scalable and maintainable enterprise applications. However, we bet you have also suffered the pain. You need to implement the getters/setters, override `equals`, `hashCode`, `toString` and/or add a constructor, for each and every single class. It makes the code cumbersome and noisy. Do you know we can actually use 1 single line as the alternative? Time to try Lombok!

Java is a very standardized programming language that always follows rules, meaning sometimes a very tiny functionality requires a lot of code. People complain about the lengthy code in Java, and the goal of Lombok is exactly to resolve such problem in a way to largely reduce the lengthy code with a simple set of annotations. Project Lombok, which can be integrated into IDEs, is able to inject code that is immediately available to the developer.

Install Lombok

This section explains how to integrate Lombok with the Apache Maven build tool. To include Lombok as a `provided` dependency, add it to your `<dependencies>` block in pom file as follows:

```
<dependencies>
  <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.16</version>
      <scope>provided</scope>
  </dependency>
</dependencies>
```

Data Annotation @Data

`@Data` is a convenient shortcut annotation that bundles the features of `@ToString`, `@EqualsAndHashCode`, `@Getter/@Setter` and `@RequiredArgsConstructor` together: In other words, `@Data` generates all the boilerplate that is normally associated with simple POJOs (Plain Old Java Objects) and beans: getters for all fields, setters for all non-final fields, and appropriate toString, equals and hashCode implementations that involve the fields of the class, and a constructor that initializes all final fields, as well as all non-final fields with no initializer that have been marked with `@NonNull`, in order to ensure the field is never null.

Let's take an example!

With Lombok

```
import lombok.AccessLevel;
import lombok.Setter;
import lombok.Data;
import lombok.ToString;

@Data public class DataExample {
  private final String name;
  @Setter(AccessLevel.PACKAGE) private int age;
  private double score;
  private String[] tags;

  @ToString(includeFieldNames=true)
  @Data(staticConstructor="of")
  public static class Exercise<T> {
    private final String name;
    private final T value;
  }
}
```

Vanilla Java

```
import java.util.Arrays;

public class DataExample {
  private final String name;
  private int age;
  private double score;
  private String[] tags;

  public DataExample(String name) {
    this.name = name;
  }

  public String getName() {
    return this.name;
  }

  void setAge(int age) {
    this.age = age;
  }

  public int getAge() {
    return this.age;
  }

  public void setScore(double score) {
    this.score = score;
  }

  public double getScore() {
    return this.score;
  }

  public String[] getTags() {
    return this.tags;
  }

  public void setTags(String[] tags) {
    this.tags = tags;
  }

  @Override public String toString() {
    verbose code
  }

  protected boolean canEqual(Object other) {
    return other instanceof DataExample;
  }

  @Override public boolean equals(Object o) {
    verbose code
  }

  @Override public int hashCode() {
    verbose code
  }
```

```
  public static class Exercise<T> {
    private final String name;
    private final T value;

    private Exercise(String name, T value) {
      this.name = name;
      this.value = value;
    }

    public static <T> Exercise<T> of(String name, T value) {
      return new Exercise<T>(name, value);
    }

    public String getName() {
      return this.name;
    }

    public T getValue() {
      return this.value;
    }

    @Override public String toString() {
      return "Exercise(name=" + this.getName() + ", value=" + this.getValue() +
")";
    }

    protected boolean canEqual(Object other) {
      return other instanceof Exercise;
    }

    @Override public boolean equals(Object o) {
      verbose code
    }

    @Override public int hashCode() {
      verbose code
    }
  }
}
```

More information can found at https://projectlombok.org/

Introduction to Apache Commons

Introduction to Apache Commons

We have seen that Lombok comes in handy, and now we will introduce another Java library - Apache Commons, which is so powerful and developer-friendly. The standard Java libraries fail to provide enough methods for manipulation of its core classes. Apache Commons Lang provides these extra methods.

Apache Commons Lang provides a lot of classes that comes in handy. In Java, especially when you manipulate strings, you may need to write much complex code to do the manipulation, but Apache Commons Lang can make this much easier, by providing many powerful methods. Moreover, overriding methods such as hashCode, toString and equals, can be much easier and simpler with Apache Commons Lang.

Here we will talk about StringUtils and FileUtils, two powerful classes.

Install Apache Commons

Similarly to Lombok, add this to your <dependencies> block in pom file as follows:

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.7</version>
</dependency>
<dependency>
  <groupId>commons-io</groupId>
  <artifactId>commons-io</artifactId>
  <version>2.5</version>
</dependency>
```

StringUtils

This is a class used to manipulate strings conveniently. More information regarding this class can be found at StringUtils Java API Doc (https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringUtils.html#equalsAnyIgnoreCase-java.lang.CharSequence-java.lang.CharSequence...-)

Let's take an example. Say, we have an array of strings, and we want to know if str = "cloud_computing_is_so_great" equals any of those strings in the array ignoring the case.

Vanilla Java Version

```
String str = "cloud_computing_is_so_great";
for (String s: strs) {
  if (str.equalsIgnoreCase(s)) {
    return true;
  }
}
return false;
```

With StringUtils

```
    String str = "cloud_computing_is_so_great";
return StringUtils.equalsAnyIgnoreCase(str, strs);
```

Shorter and easier! You can have a try in your IDE!

FileUtils

This is another powerful class aimed at manipulating files in Java. Say, we want to copy a folder and all its subfolders and files into another folder.

Vanilla Java Version

```
  public static void copy(File sourceLocation, File targetLocation) throws IOExc
eption {
      if (sourceLocation.isDirectory()) {
          copyDirectory(sourceLocation, targetLocation);
      } else {
          copyFile(sourceLocation, targetLocation);
      }
  }

  private static void copyDirectory(File source, File target) throws IOException
{
      if (!target.exists()) {
          target.mkdir();
      }

      for (String f : source.list()) {
          copy(new File(source, f), new File(target, f));
      }
  }

  private static void copyFile(File source, File target) throws IOException {
        InputStream in = new FileInputStream(source);
        OutputStream out = new FileOutputStream(target);

        byte[] buf = new byte[1024];
        int length;
        while ((length = in.read(buf)) > 0) {
            out.write(buf, 0, length);
        }
      in.close();
      out.close();
  }
```

Wow, this is really complex and troublesome as you can see. Let's see how we can improve the code by utilizing FileUtils.

With FileUtils

```
  public static void copy(File sourceLocation, File targetLocation) throws IOExc
eption {
    FileUtils.copyDirectory(FileUtils.getFile(sourceLocation),FileUtils.getFile
(targetLocation));
  }
```

That's it. Plain and simple. Have a try! More information regarding this class can be found at FileUtils Java API Doc (https://commons.apache.org/proper/commons-io/javadocs/api-2.5/org/apache/commons/io/FileUtils.html)

Checkstyle

Introduction to Checkstyle

Checkstyle is an open source (https://github.com/checkstyle/checkstyle) project that preforms static analysis of Java code and validates that the code complies with a set of coding standards.

Benefits of using a static analysis tool to validate code include:

- Consistent code styling across an entire project
- Standard naming conventions
- Detection of potential coding issues (other tools exist specifically for this purpose, i.e. Findbugs (https://github.com/findbugsproject/findbugs))
- Validation of imports and restricting use of unapproved packages

Using Checkstyle in a Maven Project

To incorporate Checkstyle in to your Maven Project, we can make use of the Apache Maven Checkstyle Plugin (https://maven.apache.org/plugins/maven-checkstyle-plugin/). The plugin usage (https://maven.apache.org/plugins/maven-checkstyle-plugin/usage.html) reference specifies the necessary `pom.xml` changes to check for violations as part of the build.

1. Add the plugin to the build section of the `pom.xml`, as reproduced below. This addition will run the `checkstyle:check` goal (from the Maven Checkstyle Plugin) during the `validate` phase of the build.

```
<project>
 ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <version>2.17</version>
        <executions>
          <execution>
            <id>validate</id>
            <phase>validate</phase>
            <goals>
              <goal>check</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
 ...
</project>
```

2. Execute `mvn package` to rebuild the project and collect the existing Checkstyle violations in the project.

> Information

You will notice that build has failed because there are 11 Checkstyle violations in this project.

```
$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
-----
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
-----
[INFO]
[INFO] --- maven-checkstyle-plugin:2.17:check (validate) @ my-app ---
[WARNING] File encoding has not been set, using platform encoding UTF-8,
i.e. build is platform dependent!
[INFO] There are 11 errors reported by Checkstyle 6.11.2 with sun_checks.x
ml ruleset.
[ERROR] src/main/java/com/mycompany/app/App.java:[0] (javadoc) JavadocPack
age: Missing package-info.java file.
[ERROR] src/main/java/com/mycompany/app/App.java:[7] (regexp) RegexpSingle
line: Line has trailing spaces.
[ERROR] src/main/java/com/mycompany/app/App.java:[7,1] (design) HideUtilit
yClassConstructor: Utility classes should not have a public or default con
structor.
[ERROR] src/main/java/com/mycompany/app/App.java:[8,1] (blocks) LeftCurly:
'{' at column 1 should be on the previous line.
[ERROR] src/main/java/com/mycompany/app/App.java:[9,5] (javadoc) JavadocMe
thod: Missing a Javadoc comment.
[ERROR] src/main/java/com/mycompany/app/App.java:[9,29] (whitespace) Paren
Pad: '(' is followed by whitespace.
[ERROR] src/main/java/com/mycompany/app/App.java:[9,30] (misc) FinalParame
ters: Parameter args should be final.
[ERROR] src/main/java/com/mycompany/app/App.java:[9,43] (whitespace) Paren
Pad: ')' is preceded with whitespace.
[ERROR] src/main/java/com/mycompany/app/App.java:[10,5] (blocks) LeftCurl
y: '{' at column 5 should be on the previous line.
[ERROR] src/main/java/com/mycompany/app/App.java:[11,28] (whitespace) Pare
nPad: '(' is followed by whitespace.
[ERROR] src/main/java/com/mycompany/app/App.java:[11,43] (whitespace) Pare
nPad: ')' is preceded with whitespace.
[INFO] ------------------------------------------------------------------------
-----
[INFO] BUILD FAILURE
...
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-checkstyle-p
lugin:2.17:check (validate) on project my-app: You have 11 Checkstyle viol
ations. -> [Help 1]
```

3. Fix the errors in `App.java` and create a `package-info.java` in the same directory as `App.java`.

```
$ cat src/main/java/com/mycompany/app/package-info.java
/**
 * Package info for Checktyle / Maven demo.
 */
package com.mycompany.app;


$ cat src/main/java/com/mycompany/app/App.java
package com.mycompany.app;

/**
 * Hello world!
 *
 */
public final class App {
    /**
     * Change default constructor to private for
     * Utility classes.
     */
    private App() {

    }

    /**
     * Main method for Hello Maven Application.
     *
     * @param args Command Line Arguments
     */
    public static void main(final String[] args) {
        System.out.println("Hello World!");
    }
}
```

4. After all the Checkstyle violations are corrected, running `package` will result in a successful build.

```
$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
-----
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
-----
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ my-app ---
[INFO] Building jar: .../my-app/target/my-app-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
-----
[INFO] BUILD SUCCESS
```

Custom Checkstyle Rules

Checkstyle supports custom configurations that allow the project owner to add additional rules or to modify rules in the case they are too restrictive. In this section we will demonstrate overriding an existing rule with a custom configuration. Refer to the Maven Checkstyle Plugin's custom checker config (https://maven.apache.org/plugins/maven-checkstyle-plugin/examples/custom-checker-config.html) page for adding customer checkers to a Maven project. Below is an example of adding a custom checker to modify the max line length.

1. Add the following comment to the `main` method of `App.java`.

```
// This comment is longer than the approved line length. Let's add a custo
m rule.
```

2. Re-build the project and see that the build fails because our comment is longer than 80 characters.

```
$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
-----
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
-----
[INFO]
[INFO] --- maven-checkstyle-plugin:2.17:check (validate) @ my-app ---
[WARNING] File encoding has not been set, using platform encoding UTF-8,
i.e. build is platform dependent!
[INFO] There is 1 error reported by Checkstyle 6.11.2 with sun_checks.xml
ruleset.
[ERROR] src/main/java/com/mycompany/app/App.java:[22] (sizes) LineLength:
Line is longer than 80 characters (found 89).
[INFO] ------------------------------------------------------------------------
-----
[INFO] BUILD FAILURE
```

3. Create a custom Checkstyle configuration in a new `config/` directory under the project root. This will increase the max value of LineLength to 120 characters.

```
$ cat config/checkstyle.xml
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
          "-//Puppy Crawl//DTD Check Configuration 1.3//EN"
"http://www.puppycrawl.com/dtds/configuration_1_3.dtd">

<module name="Checker">
    <module name="TreeWalker">
        <module name="LineLength">
            <property name="max" value="120"/>
        </module>
    </module>
</module>
```

4. Add a `configLocation` to the Maven Checkstyle Plugin configuration in the `pom.xml`. This will make the plugin aware of the custom configuration in `checkstyle.xml`.

```
...
<build>
    <plugins>
        <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <version>2.17</version>
        <configuration>
            <configLocation>config/checkstyle.xml</configLocation>
        </configuration>
...
```

5. Build the project and see that there is no longer a Checkstyle violation for line length.

```
$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
-----
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
-----
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ my-app ---
[INFO] Building jar: .../my-app/target/my-app-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
-----
[INFO] BUILD SUCCESS
```

More information on the configurations supported by Checkstyle can be found on Checkstyle's Sourceforge (http://checkstyle.sourceforge.net/config.html) page.

References

- Google Java style guide (https://google.github.io/styleguide/javaguide.html)
- Checkstyle rules for Google Java style guide (https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/google_checks.xml)
- Sun Java style guide (https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/sun_checks.xml)
- Checkstyle custom config on Sourceforge (http://checkstyle.sourceforge.net/config.html)
- Video on checkstyle/findbugs/PMD IDEA Intellij plugin usage (https://www.youtube.com/watch?v=o4pdkgHfQS4)
- Installing the coding style settings in Intellij (https://github.com/HPI-Information-Systems/Metanome/wiki/Installing-the-google-styleguide-settings-in-intellij-and-eclipse)
- Google Java Style XML (https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/google_checks.xml)
- google-java-format for IntelliJ IDEA (https://plugins.jetbrains.com/idea/plugin/8527-google-java-format)