CS304
Software Engineering

TAN, Shin Hwei
陈馨慧

Southern University of Science and Technology

Slides adapted from https://jeroenmols.com/blog/2016/03/07/resourcenaming/

Java Naming Convention

From https://www.slideshare.net/MukeshKumar411/java-naming-conventions?from_action=save

Java Naming conventions

- class name: It should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
- interface name: It should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
- method name: It should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.

- variable name: It should start with lowercase letter e.g. firstName, orderNumber etc.

- package name: It should be in lowercase letter e.g. java, lang, sql, util etc.

- constants name: It should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

```
class Student
{
    int id;
    String name;
    public static void main(String args[])
   {
     Student s1=new Student();
     System.out.println(s1.id);
     System.out.println(s1.name);
   }
}
```

```
class Student{
    int id;
    String name;
}
class TestStudent1{
   public static void main(String
    args[]){  Student s1=new Student();
    System.out.println(s1.id);
    System.out.println(s1.name);
   }}
```

```
class Student{
    int rollno;
    String name;
    void insertRecord(int r, String
     n){  rollno=r;
     name=n;
     }
void displayInformation()
{
System.out.println(rollno+" "+name);}
}
```

```
class TestStudent4{
public static void main(String
 args[]){  Student s1=new Student();
 Student s2=new Student();
 s1.insertRecord(111,"Karan");
 s2.insertRecord(222,"Aryan");
 s1.displayInformation();
 s2.displayInformation();
 }
}
```

```
class Employee{
        int id;
        String name;
        float salary;
        void insert(int i, String n, float s)
            {  id=i;
            name=n;
            salary=s;
        }
        void display()
    {System.out.println(id+" "+name+" "+salary);}
 }
```

```
public class TestEmployee {
 public static void main(String[] args) {
     Employee e1=new Employee();
     Employee e2=new Employee();
     Employee e3=new Employee();
     e1.insert(101,"ajeet",45000);
     e2.insert(102,"irfan",25000);
     e3.insert(103,"nakul",55000);
     e1.display();
     e2.display();
     e3.display();  }  }
```

```
class Rectangle
{
    int length;
    int width;
    void insert(int l, int
     w){  length=l;
     width=w;
    }
    void calculateArea()
   {System.out.println(length*width);}  }
```

```
class TestRectangle1
{
    public static void main(String
     args[]){  Rectangle r1=new
     Rectangle();  Rectangle r2=new
     Rectangle();  r1.insert(11,5);
     r2.insert(3,15);
     r1.calculateArea();
     r2.calculateArea();
 } }
```

Constructor in Java

- **Constructor in java** is a special type of method that is used to initialized the object

- Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

There are basically two rules defined for the constructor.

- Constructor name must be same as its class name
- Constructor must have no explicit return type

Types of java constructors

- There are two types of constructors:
  - Default constructor (no-arg constructor)
  - Parameterized constructor

Example of default constructor

```
class Bike1
{
 Bike1()
{
System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}
}
```

```
class Student3{
int id;
String name;
void display()
{System.out.println(id+" "+name);}
 public static void main(String args[]){
    Student3 s1=new Student3();
    Student3 s2=new Student3();
    s1.display();
    s2.display();
}}
```

Android Naming Convention

-Some groups have selected their projects to be open-source Android apps

Basic Principle fo Android Resource Naming

<WHAT>_<WHERE>_<DESCRIPTION>_<SIZE>

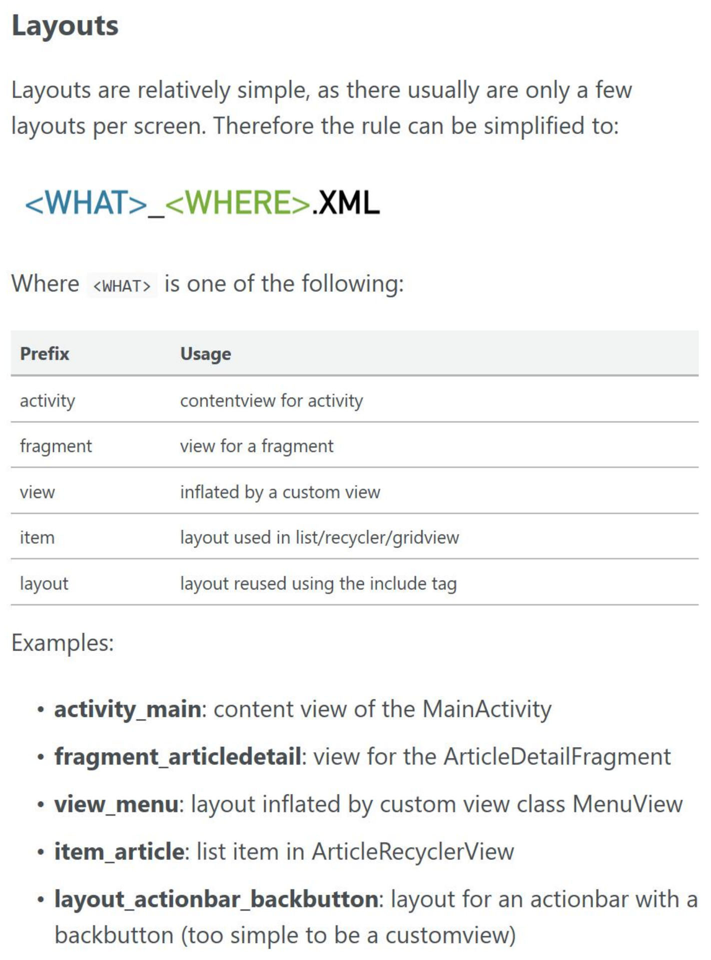
- **<WHAT>**
- Indicate what the resource actually represents, often a standard Android view class. Limited options per resource type.
- (e.g. MainActivity -> activity)

- **<WHERE>**
- Describe where it logically belongs in the app. Resources used in multiple screens use all, all others use the custom part of the Android view subclass they are in.
- (e.g. MainActivity -> main, ArticleDetailFragment -> articledetail)

- **<DESCRIPTION>**
- Differentiate multiple elements in one screen.
- (e.g. title)

- **<SIZE>** (optional)
- Either a precise size or size bucket. Optionally used for drawables and dimensions.
- (e.g. 24dp, small))

Naming Convention for Layouts



Layouts

Layouts are relatively simple, as there usually are only a few layouts per screen. Therefore the rule can be simplified to:

<WHAT>_<WHERE>.XML

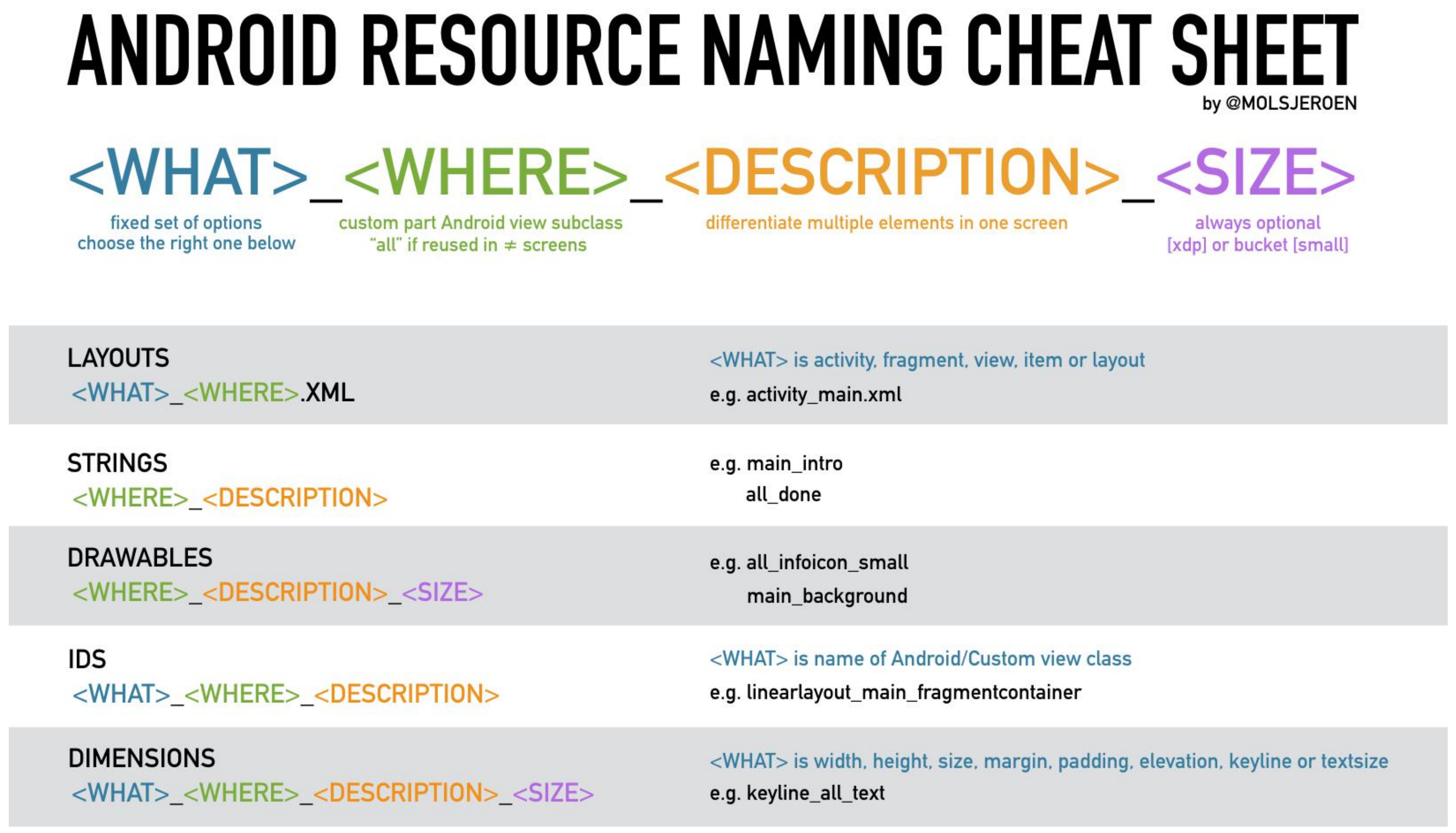
Where `<WHAT>` is one of the following:

Prefix	Usage
activity	contentview for activity
fragment	view for a fragment
view	inflated by a custom view
item	layout used in list/recycler/gridview
layout	layout reused using the include tag

Examples:

- **activity_main**: content view of the MainActivity
- **fragment_articledetail**: view for the ArticleDetailFragment
- **view_menu**: layout inflated by custom view class MenuView
- **item_article**: list item in ArticleRecyclerView
- **layout_actionbar_backbutton**: layout for an actionbar with a backbutton (too simple to be a customview)

A successful XML naming convention



ANDROID RESOURCE NAMING CHEAT SHEET

by @MOLSJEROEN

<WHAT>_<WHERE>_<DESCRIPTION>_<SIZE>

<WHAT>	fixed set of options choose the right one below
<WHERE>	custom part Android view subclass "all" if reused in ≠ screens
<DESCRIPTION>	differentiate multiple elements in one screen
<SIZE>	always optional [xdp] or bucket [small]

LAYOUTS <WHAT>_<WHERE>.XML	<WHAT> is activity, fragment, view, item or layout e.g. activity_main.xml
STRINGS <WHERE>_<DESCRIPTION>	e.g. main_intro all_done
DRAWABLES <WHERE>_<DESCRIPTION>_<SIZE>	e.g. all_infoicon_small main_background
IDS <WHAT>_<WHERE>_<DESCRIPTION>	<WHAT> is name of Android/Custom view class e.g. linearlayout_main_fragmentcontainer
DIMENSIONS <WHAT>_<WHERE>_<DESCRIPTION>_<SIZE>	<WHAT> is width, height, size, margin, padding, elevation, keyline or textsize e.g. keyline_all_text

Advantages

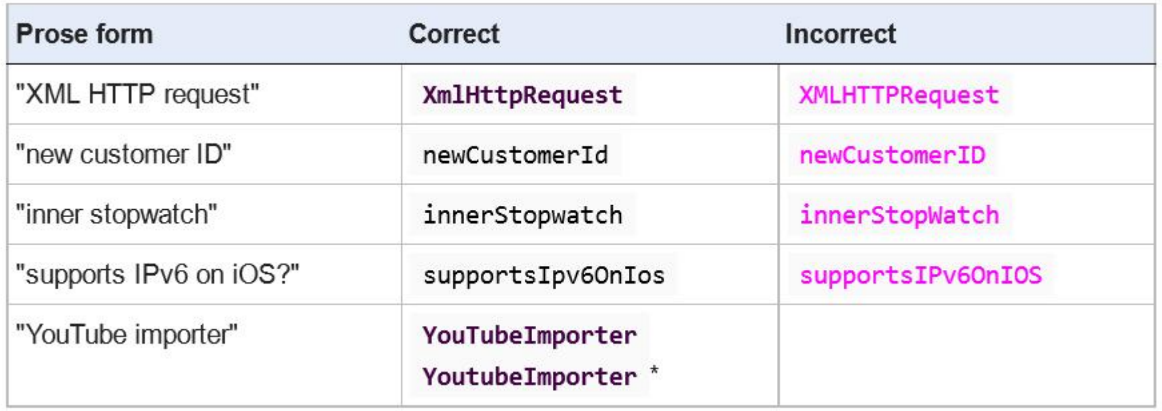
- Ordering of resources by screen
  - The *WHERE* part describes what screen a resource belongs to. Hence it is easy to get all IDs, drawables, dimensions,… for a particular screen.
- Strongly typed resource IDs
  - For resource IDs, the WHAT describes the class name of the xml element it belongs to. This makes is easy to what to cast your *findViewById()* calls to.
- Better resource organizing
  - File browsers/project navigator usually sort files alphabetically. This means layouts and drawables are grouped by their *WHAT* (activity, fragment,..) and *WHERE* prefix respectively. A simple Android Studio plugin/feature can now display these resources as if they were in their own folder.
- More efficient autocomplete
  - Because resource names are far more predictable, using the IDE's autocomplete becomes even easier. Usually entering the *WHAT* or *WHERE* is sufficient to narrow autocomplete down to a limited set of options.
- No more name conflicts
  - Similar resources in different screens are either all or have a different *WHERE*. A fixed naming scheme avoids all naming collisions.
- Cleaner resource names
  - Overall all resources will be named more logical, causing a cleaner Android project.
- Tools support
  - This naming scheme could be easily supported by the Android Studio offering features such as: lint rules to enforce these names, refactoring support when you change a *WHAT* or *WHERE*, better resource visualisation in project view,…

Google Java Style Guide

Adapted from
https://jntuabookblog.files.wordpress.com/2017/02/google-java-style-guide.pdf (English)
https://blog.csdn.net/Harrytsz/article/details/63328086 (Chinese)
Check google-java-style-guide.pdf

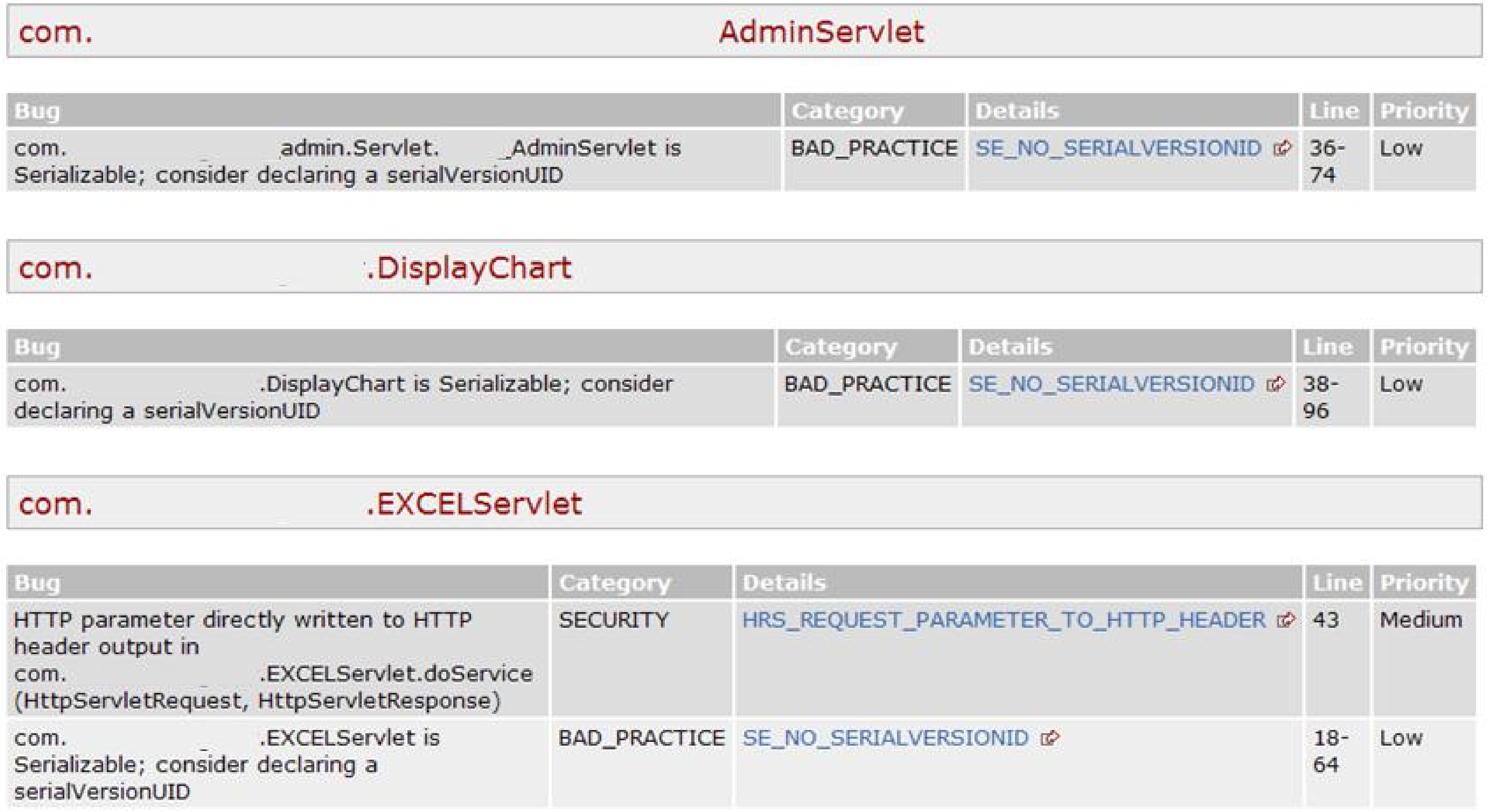
Camel case

- Class names
  - UpperCamelCase.
  - Ex. Character/ImmutableList
  - Test classes are named starting with the name of the class they are testing, and endin g with Test . For example, HashTest or HashIntegrationTest .
- Method names
  - lowerCamelCase
  - Ex. sendMessage
  - The casing of the original words is disregarded (完全忽略)



Prose form	Correct	Incorrect
"XML HTTP request"	`XmlHttpRequest`	`XMLHTTPRequest`
"new customer ID"	`newCustomerId`	`newCustomerID`
"inner stopwatch"	`innerStopwatch`	`innerStopWatch`
"supports IPv6 on iOS?"	`supportsIpv6OnIos`	`supportsIPv6OnIOS`
"YouTube importer"	`YouTubeImporter` `YoutubeImporter` *	

FindBugs Detail



com. AdminServlet

Bug	Category	Details	Line	Priority
com. admin.Servlet. AdminServlet is Serializable; consider declaring a serialVersionUID	BAD_PRACTICE	SE_NO_SERIALVERSIONID	36-74	Low

com. .DisplayChart

Bug	Category	Details	Line	Priority
com. .DisplayChart is Serializable; consider declaring a serialVersionUID	BAD_PRACTICE	SE_NO_SERIALVERSIONID	38-96	Low

com. .EXCELServlet

Bug	Category	Details	Line	Priority
HTTP parameter directly written to HTTP header output in com. .EXCELServlet.doService (HttpServletRequest, HttpServletResponse)	SECURITY	HRS_REQUEST_PARAMETER_TO_HTTP_HEADER	43	Medium
com. .EXCELServlet is Serializable; consider declaring a serialVersionUID	BAD_PRACTICE	SE_NO_SERIALVERSIONID	18-64	Low

PMD

PMD scans Java source code and looks for potential problems like:

- Possible bugs - empty try/catch/finally/switch statements
- Dead code - unused local variables, parameters and private methods
- Suboptimal code - wasteful String/StringBuffer usage
- Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
- Duplicate code - copied/pasted code means copied/pasted bugs

PMD RuleSets

- Android Rules: These rules deal with the Android SDK.
- Basic JSF rules: Rules concerning basic JSF guidelines.
- Basic JSP rules: Rules concerning basic JSP guidelines.
- Basic Rules: The Basic Ruleset contains a collection of good practices which everyone should follow.
- Braces Rules: The Braces Ruleset contains a collection of braces rules.
- Clone Implementation Rules: The Clone Implementation ruleset contains a collection of rules that find questionable usages of the clone() method.
- Code Size Rules: The Code Size Ruleset contains a collection of rules that find code size related problems.
- Controversial Rules: The Controversial Ruleset contains rules that, for whatever reason, are considered controversial.
- Coupling Rules: These are rules which find instances of high or inappropriate coupling between objects and packages.
- Design Rules: The Design Ruleset contains a collection of rules that find questionable designs.
- Import Statement Rules: These rules deal with different problems that can occur with a class' import statements.
- J2EE Rules: These are rules for J2EE
- JavaBean Rules: The JavaBeans Ruleset catches instances of bean rules not being followed.
- JUnit Rules: These rules deal with different problems that can occur with JUnit tests.
- Jakarta Commons Logging Rules: Logging ruleset contains a collection of rules that find questionable usages.
- Java Logging Rules: The Java Logging ruleset contains a collection of rules that find questionable usages of the logger.
- Migration Rules: Contains rules about migrating from one JDK version to another.
- Migration15: Contains rules for migrating to JDK 1.5
- Naming Rules: The Naming Ruleset contains a collection of rules about names - too long, too short, and so forth.
- Optimization Rules: These rules deal with different optimizations that generally apply to performance best practices.
- Strict Exception Rules: These rules provide some strict guidelines about throwing and catching exceptions.
- String and StringBuffer Rules: Problems that can occur with manipulation of the class String or StringBuffer.
- Security Code Guidelines: These rules check the security guidelines from Sun.
- Type Resolution Rules: These are rules which resolve java Class files for comparisson, as opposed to a String
- Unused Code Rules: The Unused Code Ruleset contains a collection of rules that find unused code.

PMD Rule Example

PMD Basic Rules

- **EmptyCatchBlock:** Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported.
- **EmptyIfStmt:** Empty If Statement finds instances where a condition is checked but nothing is done about it.
- **EmptyWhileStmt:** Empty While Statement finds all instances where a while statement does nothing. If it is a timing loop, then you should use Thread.sleep() for it; if it's a while loop that does a lot in the exit expression, rewrite it to make it clearer.
- **EmptyTryBlock:** Avoid empty try blocks - what's the point?
- **EmptyFinallyBlock:** Avoid empty finally blocks - these can be deleted.
- **EmptySwitchStatements:** Avoid empty switch statements.
- **JumbledIncrementer:** Avoid jumbled loop incrementers - it's usually a mistake, and it's confusing even if it's what's intended.
- **ForLoopShouldBeWhileLoop:** Some for loops can be simplified to while loops - this makes them more concise.

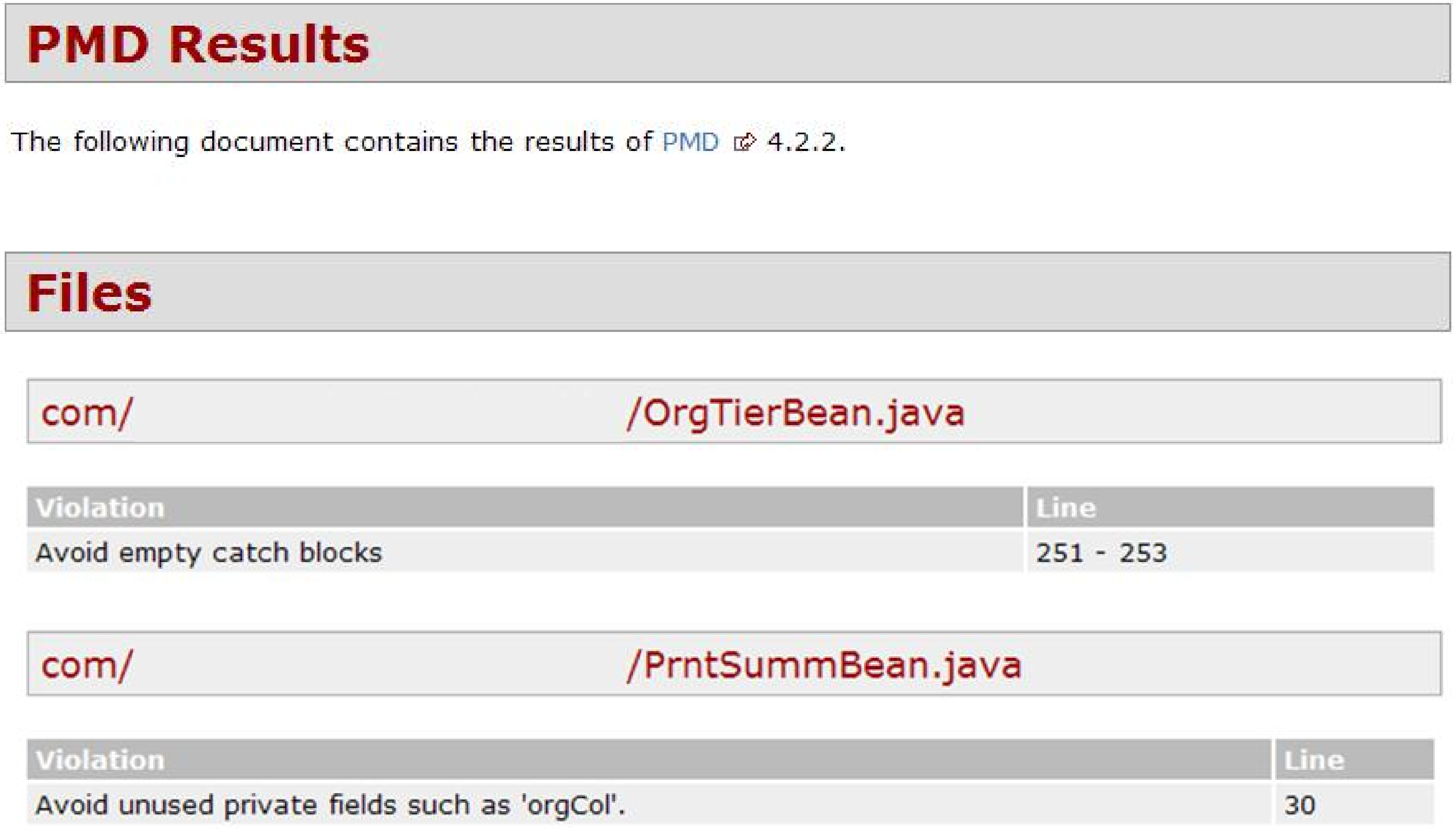
Maven PMD Configuration

```
<project>
 ...
   <reporting>
      <plugins>
        <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-pmd-plugin</artifactId>
        </plugin>
     </plugins>
   </reporting>
 ...
</project>
```

PMD Configuration

```
<reporting>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-pmd-plugin</artifactId>
            <configuration>
                <rulesets>
                    <ruleset>/rulesets/braces.xml</ruleset>
                    <ruleset>/rulesets/naming.xml</ruleset>
                    <ruleset>d:\rulesets\strings.xml</ruleset>
                    <ruleset>http://localhost/design.xml</ruleset>
                </rulesets>
            </configuration>
        </plugin>
    </plugins>
</reporting>
```

PMD Example Report



PMD Results

The following document contains the results of PMD 4.2.2.

Files

com/ /OrgTierBean.java

Violation	Line
Avoid empty catch blocks	251 - 253

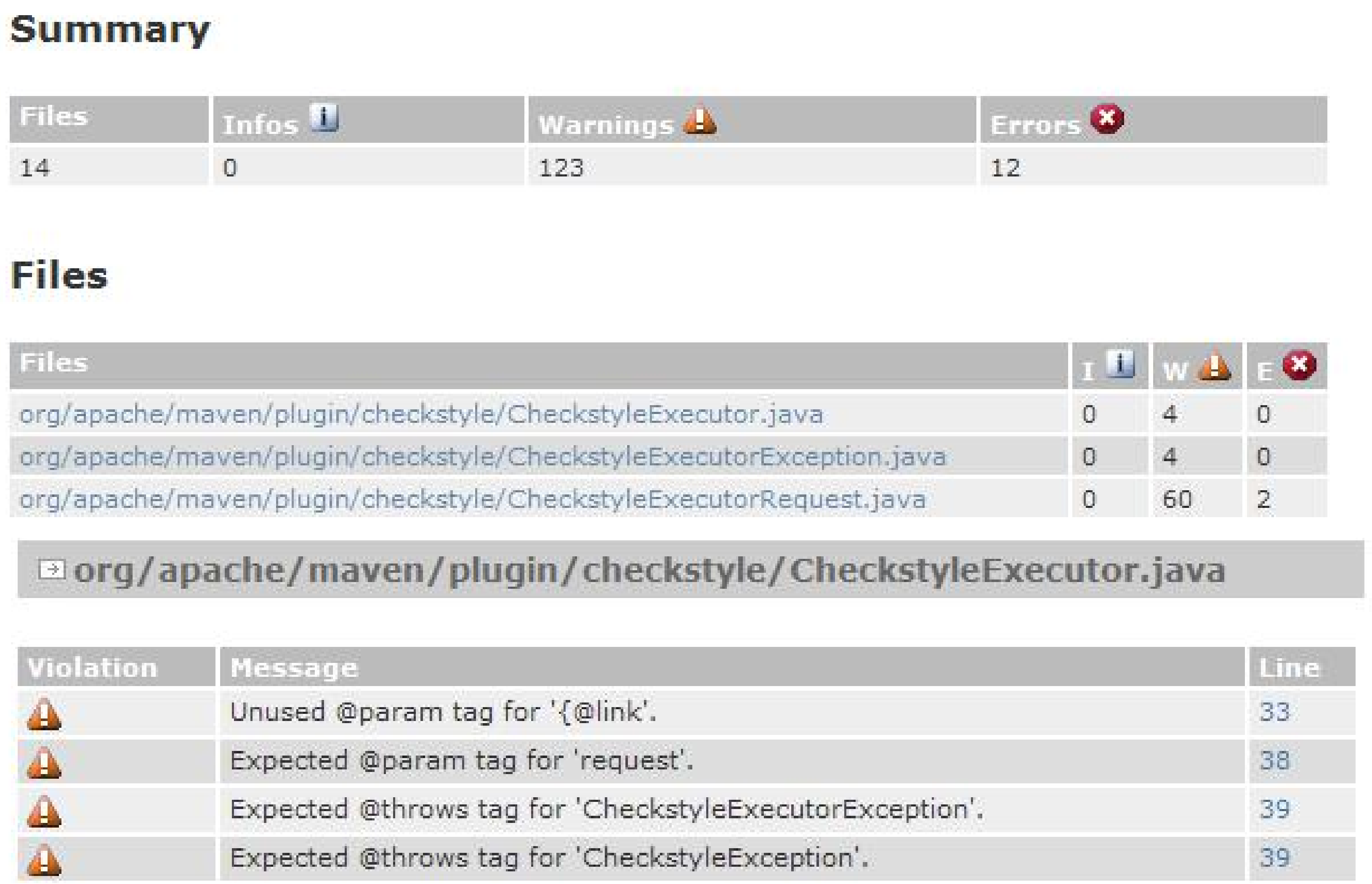
com/ /PrntSummBean.java

Violation	Line
Avoid unused private fields such as 'orgCol'.	30

CheckStyle

- Development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task.
- Highly configurable and can be made to support almost any coding standard. An example configuration file is supplied supporting the Sun Code Conventions. Other sample configuration files are supplied for other well known conventions.

CheckStyle Example



Summary

Files	Infos	Warnings	Errors
14	0	123	12

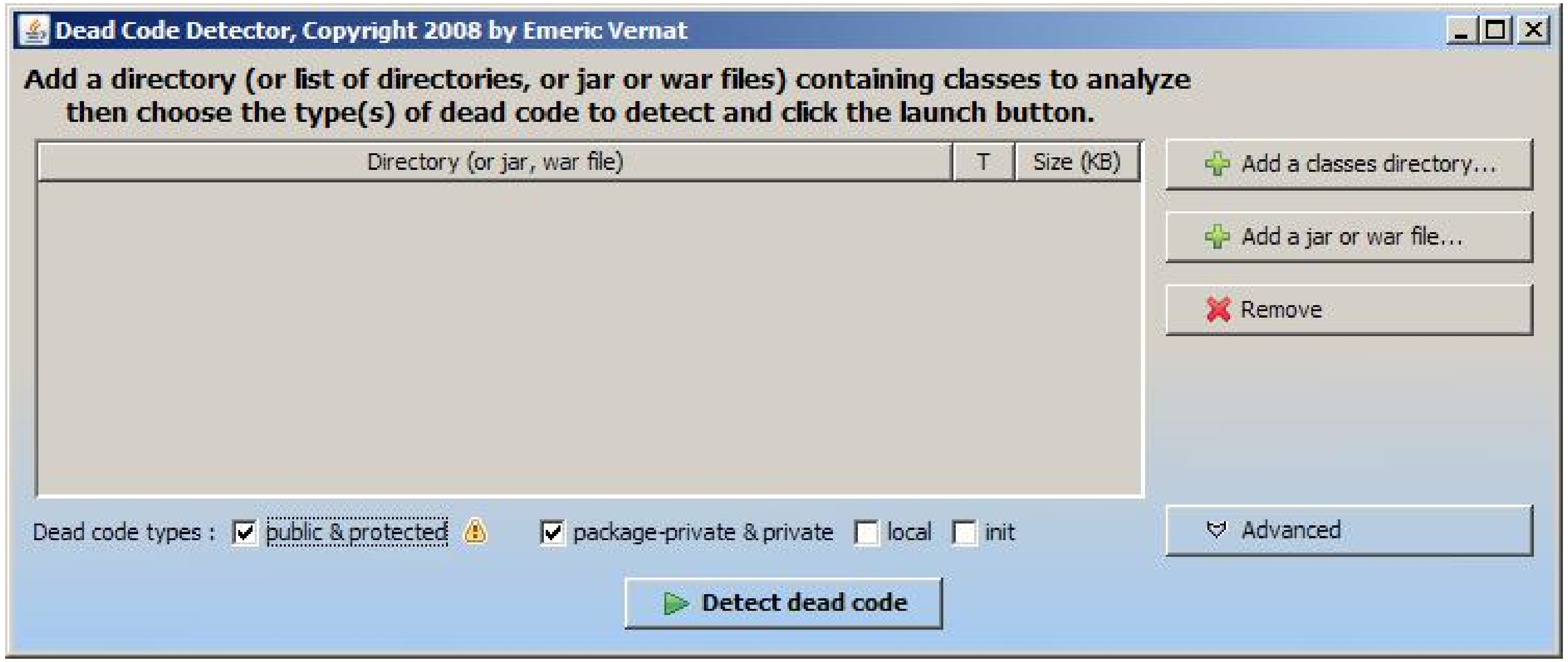
Files

Files	I	W	E
org/apache/maven/plugin/checkstyle/CheckstyleExecutor.java	0	4	0
org/apache/maven/plugin/checkstyle/CheckstyleExecutorException.java	0	4	0
org/apache/maven/plugin/checkstyle/CheckstyleExecutorRequest.java	0	60	2

org/apache/maven/plugin/checkstyle/CheckstyleExecutor.java

Violation	Message	Line
⚠	Unused @param tag for '{@link'.	33
⚠	Expected @param tag for 'request'.	38
⚠	Expected @throws tag for 'CheckstyleExecutorException'.	39
⚠	Expected @throws tag for 'CheckstyleException'.	39

Dead Code Detector


Dead Code Detector, Copyright 2008 by Emeric Vernat

Add a directory (or list of directories, or jar or war files) containing classes to analyze then choose the type(s) of dead code to detect and click the launch button.

Directory (or jar, war file)	T	Size (KB)

Dead code types : [✓ public & protected] [✓ package-private & private] local init https://code.tutsplus.com/tutorials/ensure-high-quality-android-code-with-static-analysis-tools--cms-28787

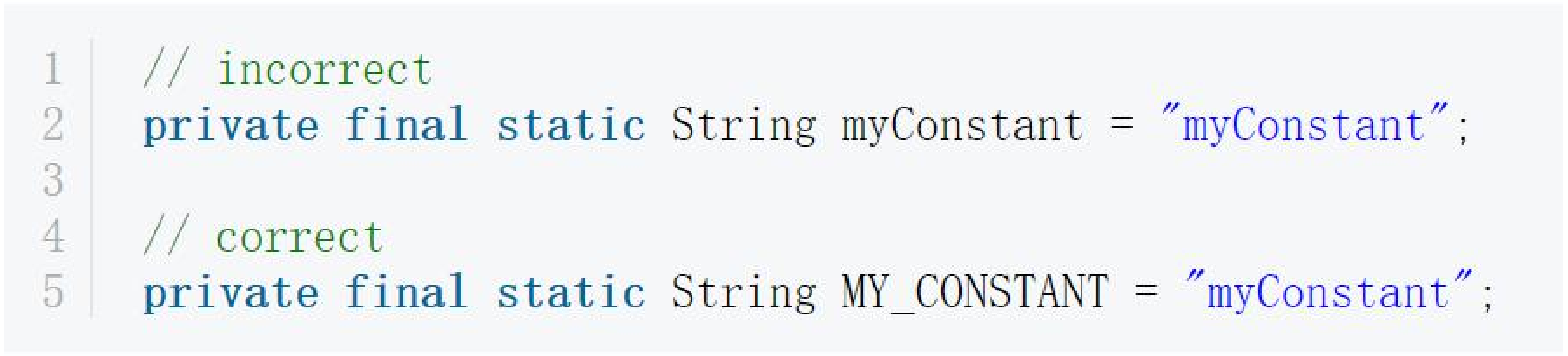
Miscellaneous Tools

- **CKJM** - Chidamber and Kemerer Java Metrics
- **Cobertura & EMMA** – Test Code Coverage
- **JavaNCSS** - A Source Measurement Suite
- **JDepend** – Package Dependencies; Efferent Couplings (Ce) (number of other packages that the classes in the package depend upon)
- **PMD-CPD** - Copy/Paste Detector (CPD)
- **Java2HTML** - Source Code turned into a colorized and browseable HTML representation.

Checkstyle

What is Checkstyle?

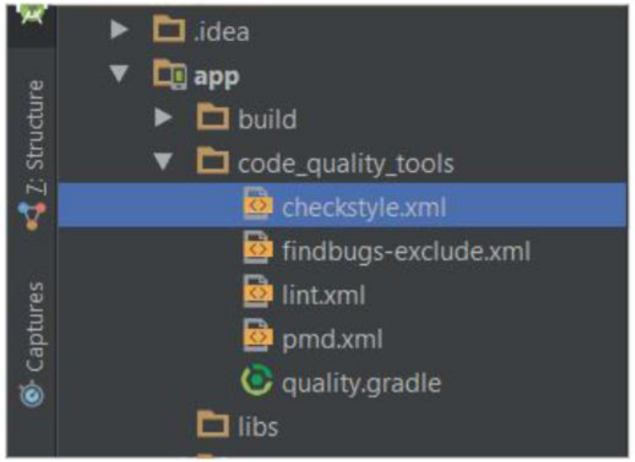
- Given rules you specify in an XML file to enforce a **coding standard** for your project, Checkstyle enforces those rules by analysing your source code and compares them against known coding standards or conventions.



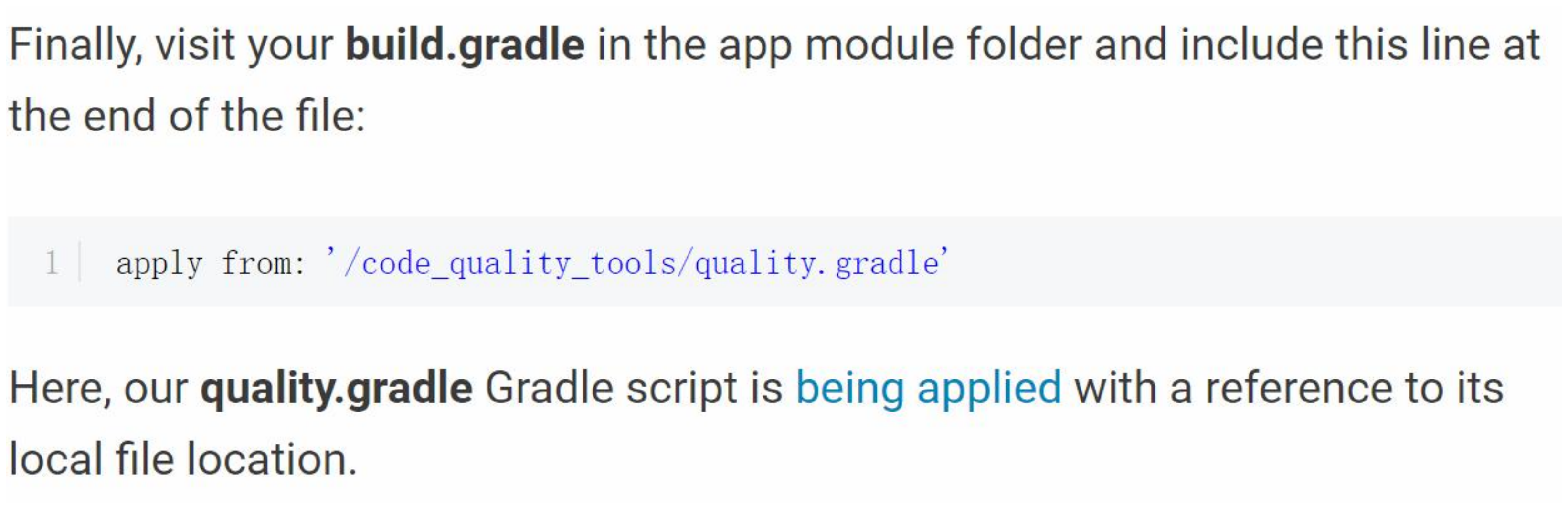
```
// incorrect
private final static String myConstant = "myConstant";

// correct
private final static String MY_CONSTANT = "myConstant";
```

Android Studio: Initial Setup

Open Android Studio and inside the app module (in **Project** view), create a new folder and name it **code_quality_tools**. This folder will contain the XML files for the code analysis tools, and it will also have a Gradle file, **quality.gradle**, which will run our static analysis tasks.



Android Studio: Initial Setup



Finally, visit your **build.gradle** in the app module folder and include this line at the end of the file:

```
apply from: '/code_quality_tools/quality.gradle'
```

Here, our **quality.gradle** Gradle script is being applied with a reference to its local file location.

How to setup Checkstyle?

Two ways:

- Install **CheckStyle-IDEA** plugin in Intellij/Android Studio (do it by yourself.)
- Install through gradle (similar to how you install evosuite through mvn)

How to setup Checkstyle for gradle?

- Create checkstyle.xml

```
<?xml version="1.0"?><!DOCTYPE module PUBLIC
    "-//Puppy Crawl//DTD Check Configuration 1.2//EN"
    "http://www.puppycrawl.com/dtds/configuration_1_2.dtd">
<module name="Checker">
  <module name="FileTabCharacter"/>
  <module name="TreeWalker">

    <!-- Checks for Naming Conventions                            -->
    <!-- See http://checkstyle.sourceforge.net/config_naming.html -->
    <module name="MethodName"/>
    <module name="ConstantName"/>

    <!-- Checks for Imports                              -->
    <!-- See http://checkstyle.sourceforge.net/config_imports.html-->
    <module name="AvoidStarImport"/>
    <module name="UnusedImports"/>

    <!-- Checks for Size                                  -->
    <!-- See http://checkstyle.sourceforge.net/config_sizes      -->
    <module name="ParameterNumber">
      <property name="max" value="6"/>
    </module>

    <!-- other rules ignored for brevity -->
</module>
```

How to setup Checkstyle for gradle?

- Create a gradle task
    - Create quality.gradle

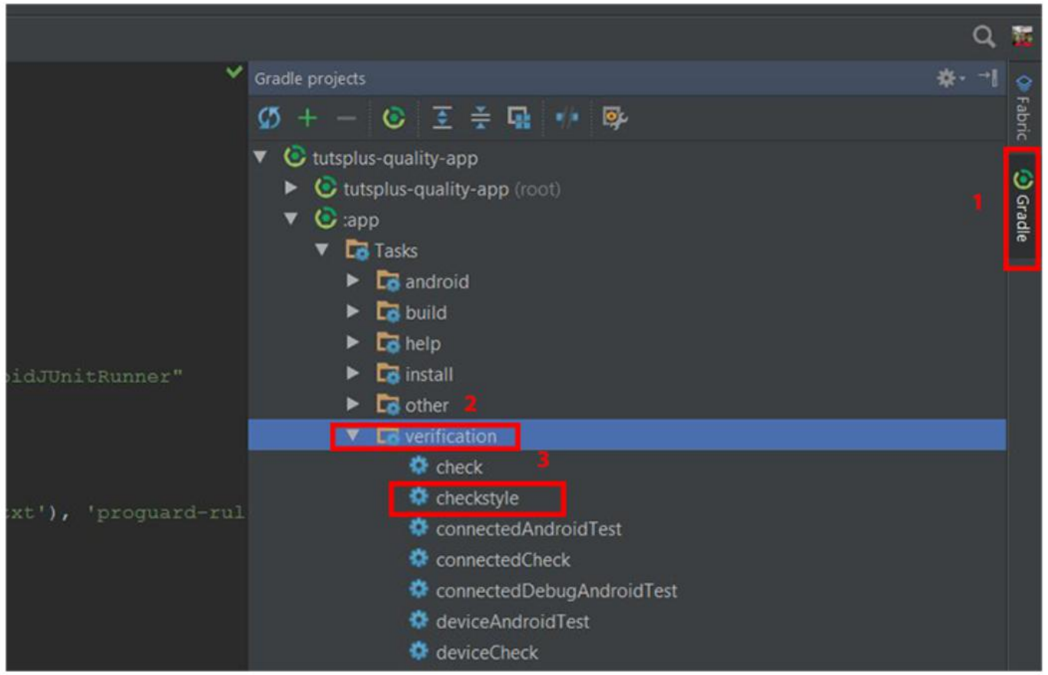
```
apply plugin: 'checkstyle'

task checkstyle(type: Checkstyle) {
    description 'Check code standard'
    group 'verification'

    configFile file('./code_quality_tools/checkstyle.xml')
    source 'src'
    include '**/*.java'
    exclude '**/gen/**'

    classpath = files()
    ignoreFailures = false
}
```

Apply checkstyle gradle pugin
configFile: the Checkstyle configuration file to use.
IgnoreFailures: whether or not to allow the build to continue if there are warnings.
include: the set of include patterns.
exclude: the set of exclude patterns. In this case, we don't scan generated classes.

How to setup Checkstyle for gradle?

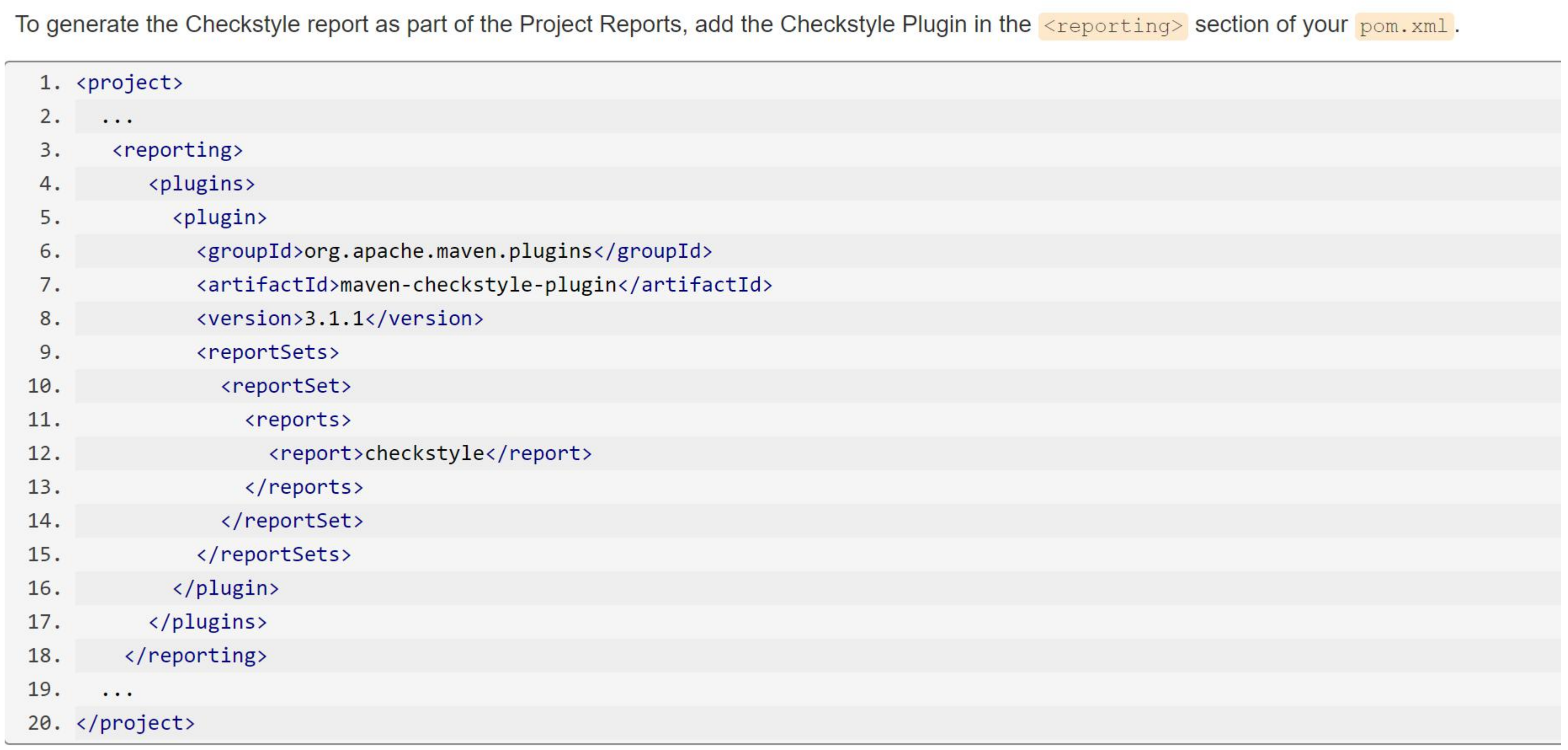
Command line: gradle checkstyle

Finally, you can run the Gradle script by visiting the Gradle tool window on Android Studio, opening the **verification** group, and then clicking on **checkstyle** to run the task.



How to setup Checkstyle for maven?

- Change pom.xml



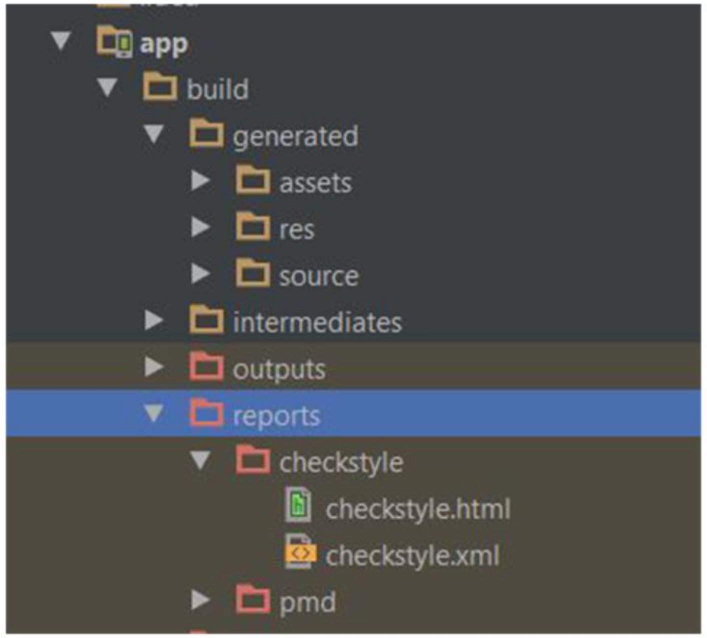
To generate the Checkstyle report as part of the Project Reports, add the Checkstyle Plugin in the `<reporting>` section of your `pom.xml`.

```
<project>
  ...
   <reporting>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-checkstyle-plugin</artifactId>
          <version>3.1.1</version>
          <reportSets>
            <reportSet>
              <reports>
                <report>checkstyle</report>
              </reports>
            </reportSet>
          </reportSets>
        </plugin>
      </plugins>
    </reporting>
  ...
</project>
```

Read: https://maven.apache.org/plugins/maven-checkstyle-plugin/usage.html

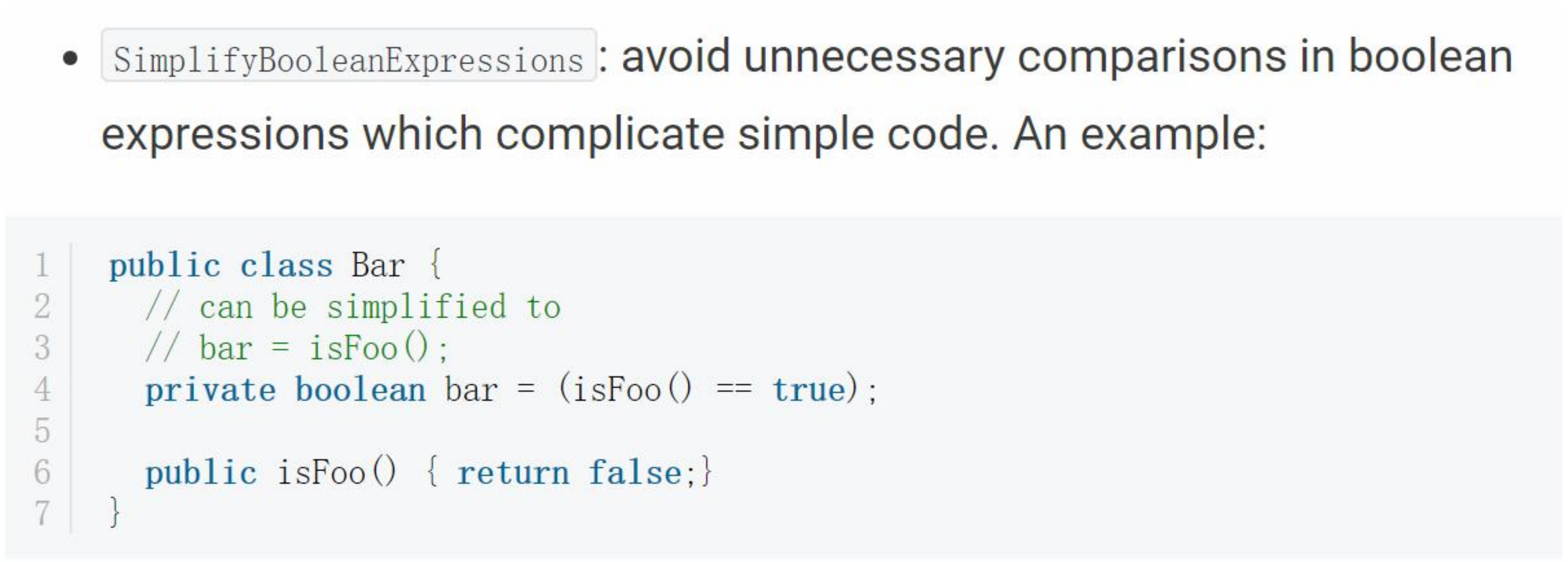
Check the report generated by Checkstyle

After the task has finished running, a report will be generated, which is available at **app module > build > reports > checkstyle**. You can open **checkstyle.html** to view the report.



What is PMD?

- Open-source code analysis tool that analyzes your source code.
- Finds common flaws like unused variables, empty catch blocks, unnecessary object creation and so on.
- PMD has many rule sets you can choose from



- `SimplifyBooleanExpressions`: avoid unnecessary comparisons in boolean expressions which complicate simple code. An example:

```java
public class Bar {
  // can be simplified to
  // bar = isFoo();
  private boolean bar = (isFoo() == true);

  public isFoo() { return false;}
}
```

How to setup PMD?

Two ways:

- Install **PMD** plugin in Intellij/Android Studio (do it by yourself.)
- Install through gradle

How to setup PMD for gradle?

- Create **pmd.xml**

```
<?xml version="1.0"?>
<ruleset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" name="Android Application Rules"
        xmlns="http://pmd.sf.net/ruleset/1.0.0"
        xsi:noNamespaceSchemaLocation="http://pmd.sf.net/ruleset_xml_schema.xsd"
        xsi:schemaLocation="http://pmd.sf.net/ruleset/1.0.0 http://pmd.sf.net/ruleset_xml_schema.xsd">

    <description>Custom ruleset for Android application</description>

    <exclude-pattern>.*/R.java</exclude-pattern>
    <exclude-pattern>.*/gen/.*</exclude-pattern>

    <!-- Android --->
    <!-- http://pmd.sourceforge.net/pmd-4.3.0/rules/android.html -->
    <rule ref="rulesets/java/android.xml"/>

    <!-- Design -->
    <!-- http://pmd.sourceforge.net/pmd-4.3.0/rules/design.html -->
    <rule ref="rulesets/java/design.xml">
        <exclude name="UncommentedEmptyMethod"/>
    </rule>

    <!-- Naming -->
    <!-- http://pmd.sourceforge.net/pmd-4.3.0/rules/naming.html -->
    <rule ref="rulesets/java/naming.xml/ShortClassName">
        <properties>
            <property name="minimum" value="3"/>
        </properties>
    </rule>
    <!-- other rules ignored for brevity -->
</ruleset>
```

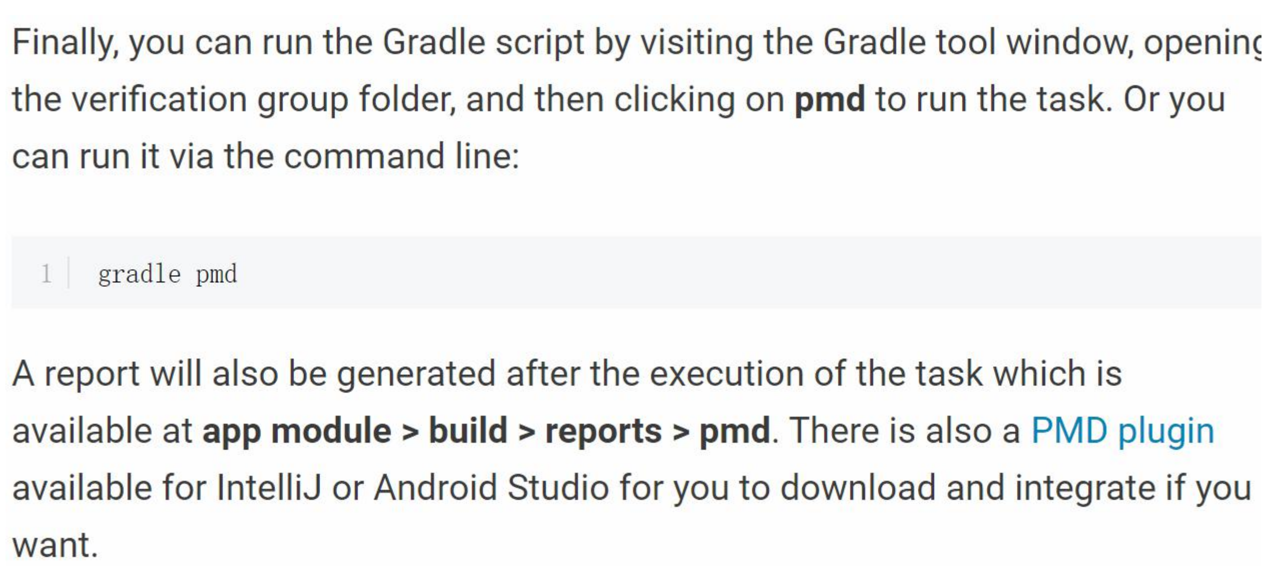
How to setup PMD for gradle?

- Create PMD Gradle task for the check to be executed inside the **quality.gradle** file.

```
apply plugin: 'pmd'

task pmd(type: Pmd) {
    description 'Run PMD'
    group 'verification'
    ruleSetFiles = files("./code_quality_tools/pmd.xml")
    source 'src'
    include '**/*.java'
    exclude '**/gen/**'
    reports {
        xml.enabled = false
        html.enabled = true
    }

    ignoreFailures = false
}
```

Apply pmd gradle pugin
ruleSetFiles: The custom rule set files to be used.
source: The source for this task.
reports: The reports to be generated by this task.
.

Run PMD for Gradle



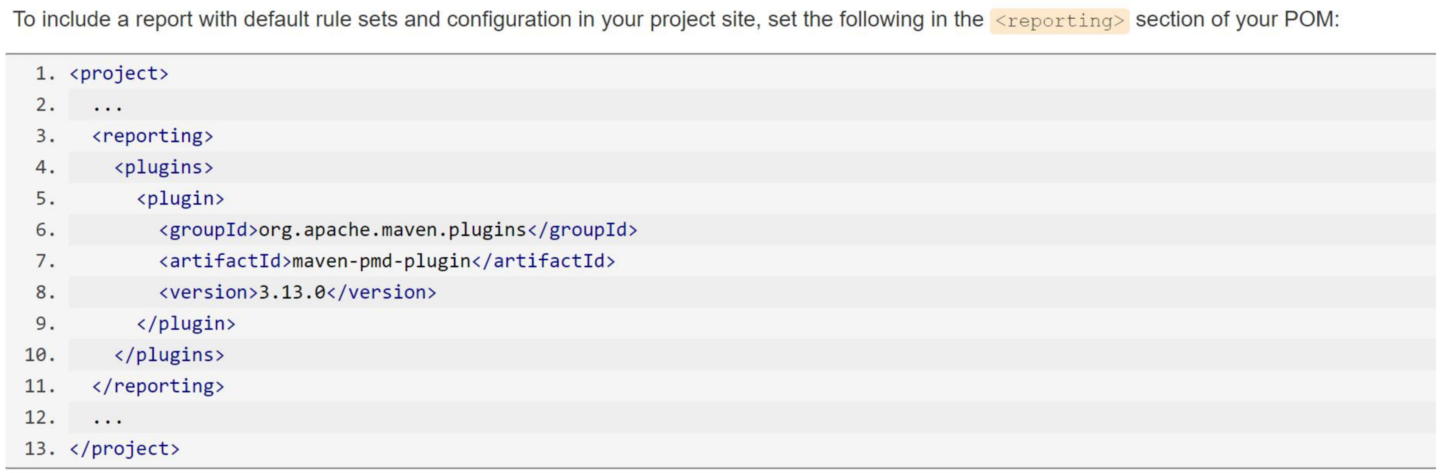
Finally, you can run the Gradle script by visiting the Gradle tool window, openin the verification group folder, and then clicking on **pmd** to run the task. Or you can run it via the command line:

```
gradle pmd
```

A report will also be generated after the execution of the task which is available at **app module > build > reports > pmd**. There is also a PMD plugin available for IntelliJ or Android Studio for you to download and integrate if you want.

How to setup PMD for maven?

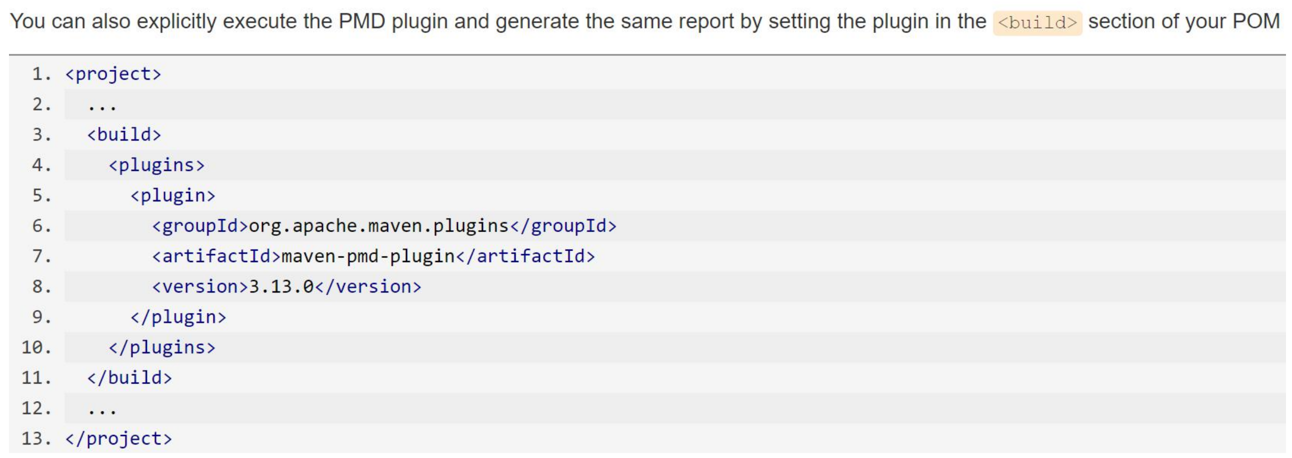
- Change pom.xml



To include a report with default rule sets and configuration in your project site, set the following in the `<reporting>` section of your POM:

```xml
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
        <version>3.13.0</version>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

- or:



You can also explicitly execute the PMD plugin and generate the same report by setting the plugin in the `<build>` section of your POM

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
        <version>3.13.0</version>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Read: https://maven.apache.org/plugins/maven-checkstyle-plugin/usage.html

What is FindBugs?

- A static analysis tool which analyses your class looking for potential problems by checking your bytecodes against a known list of bug patterns.
- Some of these patterns are:
  - **Class defines hashCode() but not equals():** A class implements the hashCode() method but not equals()—therefore two instances might be equal but not have the same hash codes. This falls under the bad practice category.
  - **Bad comparison of int value with long constant**: The code is comparing an int value with a long constant that is outside the range of values that can be represented as an int value. This comparison is vacuous and possibly will yield an unexpected result. This falls under the correctness category.
  - **TestCase** has no tests: class is a JUnit TestCase but has not implemented any test methods. This pattern is also under the correctness category.

SpotBugs

- SpotBugs is the spiritual successor of FindBugs, carrying on from the point where it left off with support of its community.
- https://spotbugs.github.io/#using-spotbugs
- SpotBugs can be used standalone and through several integrations, including:

- Ant
- Maven
- Gradle
- Eclipse

How to setup SpotBugs?

Two ways:

- Install SpotBugs plugin in IntelliJ/Android Studio (do it by yourself.)
- Install through gradle

How to setup SpotBugs/FindBugs for maven?

- Create **findbugs-exclude.xml** file

```
<FindBugsFilter>
    <!-- Do not check auto-generated resources classes -->
    <Match>
        <Class name="~.*R\$.*"/>
    </Match>

    <!-- Do not check auto-generated manifest classes -->
    <Match>
        <Class name="~.*Manifest\$.*"/>
    </Match>

    <!-- Do not check auto-generated classes (Dagger puts $ into class names) -->
    <Match>
        <Class name="~.*Dagger*.*"/>
    </Match>

    <!--
http://findbugs.sourceforge.net/bugDescriptions.html#ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD-->
     <Match>
        <Bug pattern="ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD" />
    </Match>
</FindBugsFilter>
```

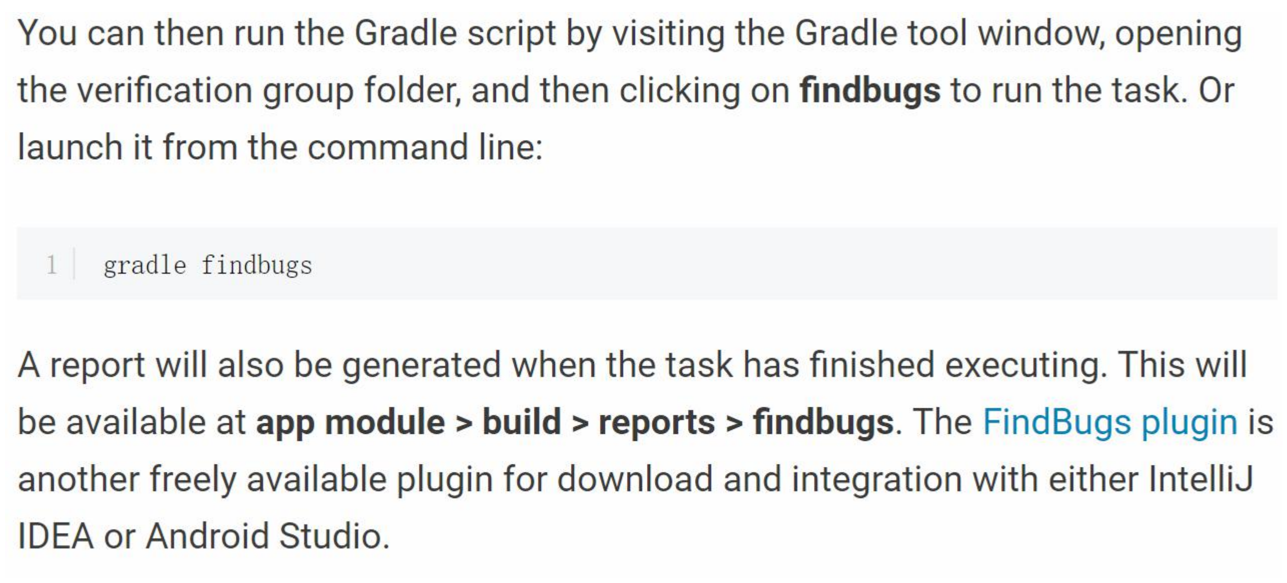
How to setup SpotBugs for gradle?

- Create PMD Gradle task for the check to be executed inside the **quality.gradle** file.

```
apply plugin: 'findbugs'

task findbugs(type: FindBugs) {
    description 'Run findbugs'
    group 'verification'
    classes = files("$project.buildDir/intermediates/classes")
    source 'src'
    classpath = files()
    effort 'max'
    reportLevel = "high"
    excludeFilter file('./code_quality_tools/findbugs-exclude.xml')
    reports {
        xml.enabled = false
        html.enabled = true
    }
    ignoreFailures = false
}
```

Apply findbugs gradle pugin
classes: the classes to be analyzed.
effort: the analysis effort level. The value specified should be one of min, default, or max. Be aware that higher levels increase precision and find more bugs at the cost of running time and memory consumption.
reportLevel: the priority threshold for reporting bugs. If set to low, all bugs are reported. If set to medium (the default), medium and high priority bugs are reported. If set to high, only high priority bugs are reported.
excludeFilter: the filename of a filter specifying bugs to exclude from being reported, which we have created already.

Run SpotBugs/FindBugs for Gradle



You can then run the Gradle script by visiting the Gradle tool window, opening the verification group folder, and then clicking on **findbugs** to run the task. Or launch it from the command line:

```
gradle findbugs
```

A report will also be generated when the task has finished executing. This will be available at **app module > build > reports > findbugs**. The FindBugs plugin is another freely available plugin for download and integration with either IntelliJ IDEA or Android Studio.

How to setup SpotBugs for Gradle?

- Change pom.xml

```
<plugin>
  <groupId>com.github.spotbugs</groupId>
  <artifactId>spotbugs-maven-plugin</artifactId>
  <version>4.0.0</version>
  <dependencies>
    <!-- overwrite dependency on spotbugs if you want to specify the version of spotbugs -->
    <dependency>
      <groupId>com.github.spotbugs</groupId>
      <artifactId>spotbugs</artifactId>
      <version>4.0.1</version>
    </dependency>
  </dependencies>
</plugin>
```

Read: https://maven.apache.org/plugins/maven-checkstyle-plugin/usage.html

What to do next?

- Integrate Checkstyle, PMD and SpotBugs into your project
  - Before each project meeting, your should run all three tools to check that there is no error left before submitting your code
  - If your project already have Checkstyle, PMD and SpotBugs integrated, make sure to run them on new code that you write
    - This will help to check if you followed their coding convention because they may have special configuration for Checktyle, PMD and Findbugs
- Nothing to submit for this lab but make sure that you know how to run static analysis for Progress Report.
- Check your score and comments for project proposal on Sakai