Improving Your Code

Dr. Isaac Griffith Idaho State University

Outcomes

After today's lecture you will be able to:

- Understand the basics of Code Style and using the Checkstyle tool to check your source code
- Reduce boilerplate and make Java programming fun again using Project Lombok
- How to check your code for common programming issues in order to make your code more professional
- How to better issue information from your system via logging

Styling Your Code

CS 2263

Why Style Your Code?

- Every programming language has its own nuances and way of being written well
- Additionally, most software companies have a documented way or style for writing code that they will consider acceptable
- Several of these style conventions have been made public for most languages.
- For Java the two most well known are
  - Sun's Java Coding Conventions (now defunct)
  - Google's Java Style Guide

Examples

- Source File Structure (Google Guide Sect. 3)
  - A source file consists of, **in order**
    1. License or copyright information, if present
    2. Package statement
    3. Import statements
    4. Exactly one top-level class
  - **Exactly one blank line** separates each section that is present
  - **Import Statements**
    - No wildcard imports
    - No line-wrapping
    - No static import for classes

Google Style Guide

- The prior example is not a complete description of section 3 as there is far more detail than can be shown in this lecture.
- Additionally there are 7 sections of the guide:
  1. Introduction
  2. Source file basics
  3. Source file structure
  4. Formatting
  5. Naming
  6. Programming Practices
  7. JavaDoc
- I would suggest that you review the guide and become familiar with it.

CheckStyle

- Because style guides tend to be quite large and contain a significant amount of information
- We need a tool which can help ensure that our code conforms to the style guide
- Thus, we have **CheckStyle**, which simply ensures that our source code conforms to a selected style guide
- Additionally, it provides us the ability to extend existing style guides, or even to create our own

Using CheckStyle

- Given that CheckStyle is a useful tool, and one that can be easily automated, there is a gradle plugin for it.
- This plugin allows us to execute CheckStyle against our source code as part of the `check` phase of the build.

build.gradle

```
plugins {
  id 'checkstyle'
}

checkstyle {
  toolVersion '8.40'
  configFile file("${rootDir}/config/checkstyle/checkstyle.xml")
}
checkstyleMain {
  source = "${projectDir}/src/main/java"
}
checkstyleTest {
  source = "${projectDir}/src/test/java"
}
```

- This plugin adds the following tasks:
  - **checkstyleMain** - runs Checkstyle against production Java source files
  - **checkstyleTest** - runs Checkstyle against test Java source files
- We can execute each of these tasks as follows:

```
$ gradle checkstyleMain
$ gradle checkstyleTest
```

Reducing Boilerplate

CS 2263

Boilerplate Code?

Idaho State University | Computer Science

- One of the arguments against Java is that it contains a large amount of structural boilerplate.
- For example let's contrast what is needed to simply print hello world in Java and in Python

Python

```python
print("Hello World")
```

Java

```java
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Boilerplate Code

- Java has been accused of requiring the following types of common boilerplate code components:
  - Getters and Setters for fields
  - Constructors which take parameters to set all fields
  - No argument constructors
  - ToString methods
  - Equals and HashCode Methods
- Java also requires some tricky code but redundant code when creating
  - Immutable classes
  - Simple Data Classes
  - Synchronization
  - Immutable Setters
  - Adding in Logging
  - Cleaning up Streams
  - Issues with requiring null checks for parameter values
- So how is it that we deal with this but retain the dependability and stability of Java?

Enter Project lombok

Project Lombok

- Project lombok is a framework designed to reduce all of the above mention issues (and several others) to a single line of code (or less)
- It does this by adding several annotations:
  - `@NonNull`
  - `@Getter/@Setter`
  - `@ToString`
  - `@EqualsAndHashCode`
  - `@NoArgsConstructor/@AllArgsConstructor/@RequiredArgsConstructor`
  - `@Data`
- We'll explore each of these in the following

@NonNull

- If on a parameter - generates a null-check in the method for you
- If on a field - will generate a null check if used in a generated method

Lombok Code

```
import lombok.NonNull;

public class NonNullExample extends Something {
 private String name;

 public NonNullExample(@NonNull Person person) {
   super("Hello");
   this.name = person.getName();
 }
}
```

Vanilla Java

```
import lombok.NonNull;

public class NonNullExample extends Something {
 private String name;

 public NonNullExample(@NonNull Person person) {
   super("Hello");
   if (person == null) {
     throw new NullPointerException("person is marked " +
                                    "@NonNull but is null");
   }
   this.name = person.getName();
 }
}
```

`@Getter` and `@Setter`

- `@Getter` generates a basic getter for the field
- `@Setter` generates a basic setter for the field

Lombok

```
import lombok.AccessLevel;
import lombok.Getter;
import lombok.Setter;

public class GetterSetterExample {

 @Getter @Setter private int age = 10;

 @Setter(AccessLevel.PROTECTED) private String name;

 @Override public String toString() {
   return String.format("%s (age: %d)", name, age);
 }
}
```

Vanilla Java

```
public class GetterSetterExample {

 private int age = 10;

 private String name;
 @Override public String toString() {
   return String.format("%s (age: %d)", name, age);
 }

 public int getAge() {
   return age;
 }

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

 protected void setName(String name) {
   this.name = name;
 }
}
```

@ToString

- Applies to a class, and generates a toString method for all included fields

Lombok

```
import lombok.ToString;

@ToString
public class ToStringExample {
 private static final int STATIC_VAR = 10;
 private String name;
 private Shape shape = new Square(5, 10);
 private String[] tags;
 @ToString.Exclude private int id;

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

 @ToString(callSuper=true, includeFieldNames=true)
 public static class Square extends Shape {
   private final int width, height;

   public Square(int width, int height) {
     this.width = width;
     this.height = height;
   }
 }
}
```

Vanilla Java

```
import java.util.Arrays;

public class ToStringExample {
 // same fields ...

 public String getName() {
   return this.name;
 }
 public static class Square extends Shape {
   private final int width, height;
   public Square(int width, int height) {
     this.width = width;
     this.height = height;
   }
   @Override public String toString() {
     return "Square(super=" + super.toString() + ", width=" + this.width +
            ", height=" + this.height + ")";
   }
 }
 @Override public String toString() {
   return "ToStringExample(" + this.getName() + ", " + this.shape + ", " +
          Arrays.deepToString(this.tags) + ")";
 }
}
```

@EqualsAndHashCode

- Generates `hashCode` and `equals` implementations from the fields of the object

Lombok

```
import lombok.EqualsAndHashCode;

@EqualsAndHashCode
public class EqualsAndHashCodeExample {
 private transient int transientVar = 10;
 private String name;
 private double score;
 @EqualsAndHashCode.Exclude private Shape shape = new Square(5, 10);
 private String[] tags;
 @EqualsAndHashCode.Exclude private int id;

 @EqualsAndHashCode(callSuper=true)
 public static class Square extends Shape {
   private final int width, height;

   public Square(int width, int height) {
     this.width = width;
     this.height = height;
   }
 }
}
```

Vanilla Java

```
public class EqualsAndHashCodeExample {
 private transient int transientVar = 10;
 private String name;
 private double score;
 private Shape shape = new Square(5, 10);
 private String[] tags;
 private int id;

 @Override public boolean equals(Object o) {
   if (o == this) return true;
   if (!(o instanceof EqualsAndHashCodeExample)) return false;
   EqualsAndHashCodeExample other = (EqualsAndHashCodeExample) o;
   if (!other.canEqual((Object)this)) return false;
   if (this.getName() == null ? other.getName() != null :
       !this.getName().equals(other.getName())) return false;
   if (Double.compare(this.score, other.score) != 0) return false;
   if (!Arrays.deepEquals(this.tags, other.tags)) return false;
   return true;
 }
```

@EqualsAndHashCode


```
@Override public int hashCode() {
  final int PRIME = 59;
  int result = 1;
  final long temp1 = Double.doubleToLongBits(this.score);
  result = (result*PRIME) + (this.name == null ? 43 :
           this.name.hashCode());
  result = (result*PRIME) + (int)(temp1 ^ (temp1 >>> 32));
  result = (result*PRIME) + Arrays.deepHashCode(this.tags);
  return result;
}

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

public static class Square extends Shape {
  private final int width, height;

  public Square(int width, int height) {
    this.width = width;
    this.height = height;
  }
```

```
  @Override public boolean equals(Object o) {
    if (o == this) return true;
    if (!(o instanceof Square)) return false;
    Square other = (Square) o;
    if (!other.canEqual((Object)this)) return false;
    if (!super.equals(o)) return false;
    if (this.width != other.width) return false;
    if (this.height != other.height) return false;
    return true;
  }

  @Override public int hashCode() {
    final int PRIME = 59;
    int result = 1;
    result = (result*PRIME) + super.hashCode();
    result = (result*PRIME) + this.width;
    result = (result*PRIME) + this.height;
    return result;
  }

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

Constructors

- `@NoArgsConstructor`
  - generates a no-args constructor
- `@AllArgsConstructor`
  - generates a constructor for all fields
- `@RequiredArgsConstructor`
  - generates a constructor for all final and non-null fields

Constructors

Idaho State University | Computer Science

Vanilla Java

```
public class ConstructorExample<T> {
 private int x, y;
 @NonNull private T description;

 private ConstructorExample(T description) {
   if (description == null) throw
        new NullPointerException("description");
   this.description = description;
 }

 public static <T> ConstructorExample<T> of(T description) {
   return new ConstructorExample<T>(description);
 }

 @java.beans.ConstructorProperties({"x", "y", "description"})
 protected ConstructorExample(int x, int y, T description) {
   if (description == null) throw
        new NullPointerException("description");
   this.x = x;
   this.y = y;
   this.description = description;
 }

  public static class NoArgsExample {
    @NonNull private String field;

    public NoArgsExample() {
    }
  }
}
```

Lombok

```
import lombok.AccessLevel;
import lombok.RequiredArgsConstructor;
import lombok.AllArgsConstructor;
import lombok.NonNull;

@RequiredArgsConstructor(staticName = "of")
@AllArgsConstructor(access = AccessLevel.PROTECTED)
public class ConstructorExample<T> {
 private int x, y;
 @NonNull private T description;

 @NoArgsConstructor
 public static class NoArgsExample {
   @NonNull private String field;
 }
}
```

@Data

- Annotates a class and is equivalent to the application of all of the following annotations:
  - `@ToString`
  - `@EqualsAndHashCode`
  - `@Getter` on all fields
  - `@Setter` on all non-final fields
  - `@RequiredArgsConstructor`

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;
 }
}
```

Additional Annotations

- `@Value` - makes immutable classes easy
- `@Builder` - makes creating a builder API for your class easy
- `@Synchronized` - provides for correct synchronization handling
- `@Cleanup` - calls `close()` methods safely with no hassle
- `@With` - creates immutable setters - methods that create a clone but with one changed field
- Additionally, there are some “experimental” annotations

Using Project Lombok

- Add the lombok gradle plugin to your `build.gradle` file in the plugins block

build.gradle

```
plugins {
    id 'io.freefair.lombok' version '5.3.0'
}
```

- You then need to activate annotation processors in your IDE
- Lombok will generate a “lombok.config” file
  - You should add the following line in it if you use Jacoco

```
lombok.addLombokGeneratedAnnotation = true
```

Checking Your Code

CS 2263

Static Analysis

- Analysis of the static components of a software system (i.e., source code, designs, etc.)
- Analyses can involve
  - Metrics
  - Reverse engineering
  - Detecting Vulnerable Code
  - Coding Errors
  - Style Issues
- There are static analysis tools for nearly every language used in commercial software development today.
- In Java two typical tools used are
  - PMD
  - SpotBugs

SpotBugs and PMD

Idaho State University | Computer Science

SpotBugs

- Tool which scans Java code for over 400 known bug types
- Bug patterns are separated into several different categories
  - Bad Practice
  - Correctness
  - Malicious code vulnerability
  - Multithreaded correctness
  - Performance
  - Security and dodgy code
  - and several other categoires
- Additionally there are several plugins
  - FindSecurityBugs - identifies 80 additional security bugs (including OWASP 10

PMD

- Scans Java source code to find potential problems
- Problems can be
  - breaking naming conventions
  - unused code or variables
  - performance and complexity of code
  - other possible bugs

Using PMD

Idaho State University | Computer Science

build.gradle

```
plugins {
  id 'pmd'
}

pmd {
  consoleOutput = true
  toolVersion = "6.21.0"
  rulesMinimumPriority = 5
  ruleSets = [
      "category/java/errorprone.xml",
      "category/java/bestpractices.xml"
  ]
}
```

- Provides several tasks, all of which run during the `check` phase of a build
  - **pmdMain** - runs PMD against Java source files
  - **pmdTest** - runs PMD against Java test files
- You can execute these tasks as follows:

```
$ gradle pmdMain
$ gradle pmdTest
```

Using Spotbugs

- SpotBugs will run as part of the `check` phase of a build
- You can execute spotbugs only using the spotbugs task as follows:

```
$ gradle spotbugs
```

build.gradle

```
plugins {
  id 'com.github.spotbugs' version '4.6.1'
}

dependencies {
  spotbugs 'com.github.spotbugs:spotbugs:4.2.1'
}
```

Logging

CS 2263

What is Logging?

- Logging provides an application the ability to record its activity
- This record can be in a variety of forms
  - Database entries
  - A streaming file
  - Console output
  - A combination of the above
- Nearly all languages provide a logging mechanism, or have a library which does

What is Logging?

- Logging mechanisms are typically subdivided into three components
  - **Logger** - responsible for capturing messages to be logged (along with metadata) and passing this information to the framework
  - **Logging Framework** - calls the formatter with the message, which then formats the message for output
  - **Appender/Handler** - after formatting the framework passes the message to this component which outputs the message to:
    - Console
    - File
    - Database
    - or Email

Logging

- Logging typically involves a setup which specifies
  - Name of the logger (typically the class name to which the logger is attached)
  - Minimum severity level, for example
    - SEVERE
    - WARNING
    - INFO
    - CONFIG
    - FINE
    - FINER
    - FINEST
  - Filters used to filter out logging levels, events, etc.
  - Formatters, layouts, or renderers
  - Handlers/Appenders

Logging Frameworks

- Currently there are many different logging frameworks
  - Built in Java Logging API
  - Apache Commons Logging (JCL)
  - log4j
  - tinylog
  - Logback
  - SLF4J
- The variety of choices has led to several issues including integration problemes and problems with using multiple loggers across projects
  - This led to the development of various wrappers to solve this problem
  - Google solved this problem by developing a better Logging Framework -> Flogger

Resources

Idaho State University | Computer Science

- JavaDoc
  - Tutorialspoint Tutorial
  - Javadoc Coding Standards
  - Tips and Tricks for Better Java Docs
- Licenses
  - choosealicense.com
  - Open Source Initiative
  - Gradle License Plugin
- Style
  - Google Java Style Guide
  - Checkstyle Gradle Plugin
  - Checkstyle Google Java Guide File
- Lombok
- Analysis
  - SpotBugs Plugin
  - PMD Plugin
- Code Coverage
  - Jacoco Plugin
  - Definitive Guide to Jacoco Gradle Plugin
- Java Best Practices
  - javapractices.com

For Next Time

- Review this lecture
- Watch Lecture 20

Are there any questions?