

# Libraries

Archie Powell

November 28th, 2023

Libraries



# Table of Contents

- 1 Libraries
- 2 Maven
- 3 Lambda Expressions
- 4 Lecture summary



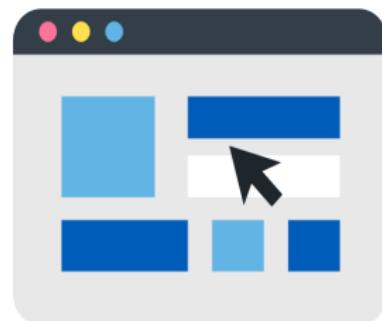
# Section 1

## Libraries



# Common functionality

- When writing Java code, it is often likely that a problem you want to solve has already been solved by another programmer elsewhere, e.g
  - Adding authentication
  - Adding database functionality
  - Adding a Graphical User Interface (GUI)
  - Adding machine learning functions
- It doesn't make sense to re-invent the wheel and implement these from scratch.
- Ideally you would want to use methods and objects that others have already written.
  - We touched a little on this when we looked at packages in Week 7.



# A summary of Java libraries

- A library is a collection of classes and methods that have already been written, available to use.
- The library is accessed through an application programming interface (API), which connects the code the user is writing with the pre-written code in the library.
- Documentation is usually in the form of a *Javadoc*, an HTML document which specifies all of the methods, classes and parameters of the library API.



# Linking to Java libraries

- Connecting your code to the library is known as *linking* to the library.
- When compiling a program on the command line, Java will look for libraries in the directory of the class you want to compile.
  - e.g. compiling with `javac MyProgram.java` will look for libraries in the directory where `MyProgram.java` lies.
- However, we might want to link to libraries in other locations.



# Environment variables and Classpath

- An environment variable is a system-wide variable that can be accessed by any program running on your operating system.
- In addition to looking in the current directory, Java will also look for libraries in directories specified in the *CLASSPATH* environment variable.
  - Many programming languages use environment variables to specify where they should look for libraries.
  - In C++, the environment variable is called *LD\_LIBRARY\_PATH*.
  - In Python, the environment variable is called *PYTHONPATH*.
  - In Java, the environment variable is called *CLASSPATH*.

# Classpath on Linux/Mac (Demo)

- To see the current value of CLASSPATH, you can open up a Terminal window and use the *echo* command:
  - echo \$CLASSPATH
- To add to the directory, you use the *export* command:
  - export CLASSPATH=/path/to/your/directory:\$CLASSPATH
- Alternatively, you can add the parameter *-cp* to the compile statement:
  - javac -cp .:path/to/first/jarFile.jar:path/to/second/jarFile.jar MyProgram.java
- If you add the parameter to the compile statement, you must also add it to the run statement.

# Classpath on Windows

- To see the current value of CLASSPATH, you can open up a Command Prompt window and use the *echo* command:
  - echo %CLASSPATH%
- To add to the directory, you use the *set* command:
  - set CLASSPATH=C:\path\to\your\directory;%CLASSPATH%
- Alternatively, you can add the parameter *-cp* to the compile statement:
  - javac -cp .;path\to\first\jarFile.jar;path\to\second\jarFile.jar MyProgram.java
- If you add the parameter to the compile statement, you must also add it to the run statement.

# Adding directories to Classpath permanently

- If you close the terminal or command prompt window, the CLASSPATH variable will reset.
- To permanently add directories on Mac/Linux, you can add the export statement to your bash configuration file (e.g., `~/.bashrc`, `~/.bash_profile`, `~/.zshrc`):
  - `echo 'export CLASSPATH=/path/to/your/directory:$CLASSPATH'`  
`>> ~/.bashrc`
- To permanently add directories on Windows, you can add the CLASSPATH variable via System Properties:
  - Right click *Computer* or *This PC* and select *Properties*
  - Click *Advanced system settings*
  - Click *Environment Variables*
  - In the *System variables* section, click on *New* and add a variable named *CLASSPATH* with the value being the path to your directory.

## Section 2

Maven



# A need for better library management

- We've seen how to link individual Java libraries to your java code using the CLASSPATH environment variable or via an IDE.
- Can anyone think of any problems when using libraries in this way?

# A need for better library management

- We've seen how to link individual Java libraries to your java code using the CLASSPATH environment variable or via an IDE.
- Can anyone think of any problems when using libraries in this way?
  - At scale it becomes difficult to keep track of all the libraries.
  - We have no way of updating libraries to more recent versions.
  - There may be dependencies shared between libraries which can cause problems.
  - We may want different libraries for different projects, requiring us to modify our CLASSPATH each time.



# Introducing Maven

- Maven is a project management and build automation tool, which addresses all of the problems that you encounter when using libraries manually.
- Within a Java project, an additional Project Object Model (POM) file will be included, called *pom.xml*.
- This file contains a description of all the libraries and dependencies of a project, which are retrieved from a central repository.



# pom.xml (Demo)

```
m pom.xml (ChartsDemoMaven) ✘ Main.java : ✓  
1  <?xml version="1.0" encoding="UTF-8"?>  
2  <project xmlns="http://maven.apache.org/POM/4.0.0"  
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0  
5      <modelVersion>4.0.0</modelVersion>  
6  
7      <groupId>org.example</groupId>  
8      <artifactId>ChartsDemoMaven</artifactId>  
9      <version>1.0-SNAPSHOT</version>  
10  
11     <properties>  
12         <maven.compiler.source>21</maven.compiler.source>  
13         <maven.compiler.target>21</maven.compiler.target>  
14         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
15     </properties>  
16  
17     <dependencies>  
18         <dependency>  
19             <groupId>org.jfree</groupId>  
20             <artifactId>jfreechart</artifactId>  
21             <version>1.5.4</version>  
22         </dependency>  
23  
24     </dependencies>  
25  
26 </project>
```



## Section 3

### Lambda Expressions



# What is a Lambda Expression and Why?

- Lambda expressions are a new feature introduced in Java 8 (March 2014).
- In essence, a lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.
- Why use lambda expressions?
  - Better and more compact code representation.
  - Better support for multi-core CPUs to process the data.
  - Allows you to implement an interface in a single line.



# Basic Syntax

- Operator: ->
  - The left side of the operator is the parameter(s), and the right side is the main body.
- A single parameter with one expression:
  - parameter -> expression
- Multiple parameters with one expression:
  - (parameter1, parameter2, ...) -> expression
- With multiple statements:
  - (parameter1, parameter2, ...) -> {statements;}

# Lambda Expression Examples (Demo)

- You can easily access other local variables within the lambda expression, but there are some conditions:
  - Within lambda expression, it is NOT allowed to declare a parameter that shares the same name with a local variable.
  - The value of the local variables referenced/used in a lambda expression cannot be changed.
- Iterating collections using Lambda Expressions:
  - Select all the students who are male and younger than 22, put them into a new List and print out their name and id from that new List.

## Section 4

Lecture summary



# Lecture summary

- 1 Libraries
- 2 Maven
- 3 Lambda Expressions
- 4 Lecture summary

## Questions

**Thank you! Questions?**

