Dependency Injection Using Google Guice

Daniel Hinojosa

Introduction to Dependency Injection

- In layman's terms, a wiring framework.
- The Art of "Componentizing" Software.
- Bringing back the promise of "Code Reuse"
- Ties into TDD (Test Driven Development)
- Decoupling Components as a better strategy
- Advantage of Testable, and Modular Code
- Adherence to the Single Responsibility Principle

The Single Responsibility Principle

> The single responsibility principle states that every module or class should have responsibility over a single part of the functionality provided by the software,

Tight Coupling As An Example

Sample from "Guide to Writing Testable Code"

```
class House {
  Kitchen kitchen = new Kitchen();
  Bedroom bedroom;

  House() {
    bedroom = new Bedroom();
  }

  // ...
}
```

http://misko.hevery.com/code-reviewers-guide/flaw-constructor-does-real-work/

Removing That Coupling As An Example

Removing the tight coupling

```
class House {
  Kitchen kitchen;
  Bedroom bedroom;

  // Have Guice create the objects
  //   and pass them in
  @Inject
  House(Kitchen k, Bedroom b) {
    kitchen = k;
    bedroom = b;
  }
  // ...
}
```

Bringing in a Heftier Example

A Real World Example

```
package com.xyzcorp;

import java.time.LocalDate;
import java.util.List;

public class AlbumService {
    private final AlbumDAO dao;

    public AlbumService(AlbumDAO dao) {
        this.dao = dao;
    }

    public List<Album> findTop10AlbumsForTheWeek
                               (LocalDate localDate) {
        return this.dao.findTopAlbums(localDate, 10);
    }
}
```

Better Design and Loose Coupling Makes Testable Software

Running a test

```
@Test
public void testGettingTheTop10List() {
    AlbumDAO albumDAO = mock(AlbumDAO.class); //Using Mockito
    LocalDate date = LocalDate.of(2016, 7, 24);
    when(albumDAO.findTopAlbums(date, 10)).thenReturn(fakeList);

    //Subject Under Test
    AlbumService albumService = new AlbumService(albumDAO);
    albumService.findTop10AlbumsForTheWeek(date);
}
```

Wiring the classic way without a framework

```
public static void main(String[] args) {
  AlbumDAO albumDAO = new MySQLDAO("jdbc:mysql://localhost:3306", "scott", "tiger");
  AlbumService albumService = AlbumService.setAlbumDAO(albumDAO);
  //Tons of boilerplate code for setup afterwards
}
```

NOTE We can plugin a `MySQLDAO` or `PostgreSQLDAO` or a `MongoDBDAO`

Downside to this method:

- Can get real messy
- Not flexible
- Not maintainable over time

Benefits of Google Guice and Dependency Injection in General

Using Google Guice

```
public class Runner {
    public static void main(String[] args) {
        Injector injector = Guice.createInjector(new AlbumModule());
        AlbumService albumService = injector.getInstance(AlbumService.class);
//Everything is already wired!
        //0 or minimum of lines after this.
    }
}
```

NOTE	DAO Code is wired in with annotations and Google Guice

Standard List Of Design Patterns

Select Essential Creational Patterns

Abstract Factory

Abstract Factory Pattern Defined

- Provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes
- When to use:
  - The client should be independent of how the products are created.
  - The application should be configured with one of multiple families of products.
  - Objects need to be created as a set, in order to be compatible.
  - You want to provide a collection of classes and you want to reveal just their contracts and their relationships, not their implementations.

Builder

Builder Pattern Defined

- To simplify complex object creation by defining a class whose purpose is to build instances of another class. The Builder produces one main product, such that there might be more than one class in the product, but there is always one main class.
- When to use:
  - Has complex internal structure (especially one with a variable set of related objects).
  - Has attributes that depend on each other. One of the things a Builder can do is enforce staged construction of a complex object. This would be required when the Product attributes

depend on one another. For instance, suppose you're building an order. You might need to ensure that you have a state set before you move on to "building" the shipping method, because the state would impact the sales tax applied to the Order itself.

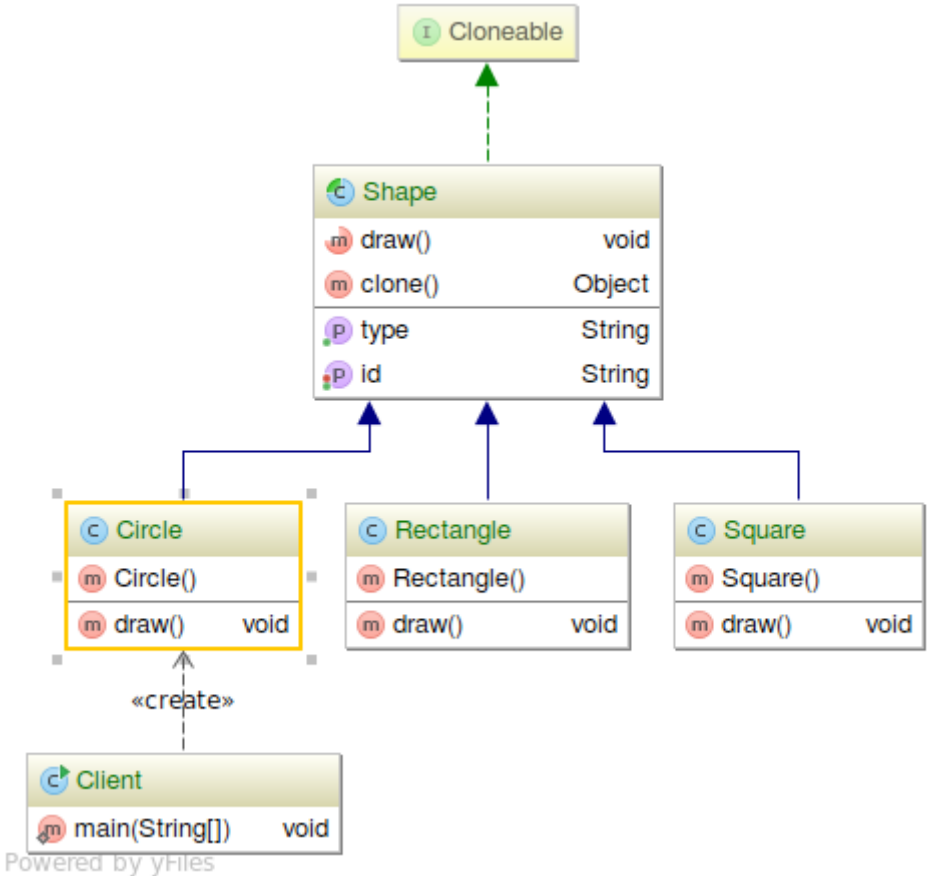
- Uses other objects in the system that might be difficult or inconvenient to obtain during creation.

Builder Use Code

Barista.java

```
EspressoDrink drink =
                EspressoDrinkMaker.addShots(2)
                   .addWhip().addSprinkles()
                   .addSkimMilk().build();
```

Prototype Pattern



Prototype Pattern Defined

- To make dynamic creation easier by defining classes whose objects can create duplicates of themselves.

Prototype Pattern in Code

```
public abstract class Shape implements Cloneable {
    private String id;
    protected String type;

    abstract void draw();

    public Object clone() {
        Object clone = null;
        try {
            clone = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
}
```

NOTE This just is just one strategy WARNING: This only performs a *shallow copy*

Singleton

Singleton Defined

- To have only one instance of this class in the system, while allowing other classes to get access to this instance.
- Without the proper control, your application will get into “thread wars.”

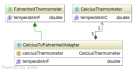
Singleton in Code

```
public class LazySingleton {
    private static LazySingleton instance = null;

    private LazySingleton() {}

    public static synchronized LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

WARNING	This pattern fell out of favor due to it creating global like variables

Select Essential Structural Patterns

Adapter Pattern



Adapter Pattern Defined

- To act as an intermediary between two classes, converting the interface of one class so that it can be used with the other.
- When to use:
  - You want to use an object in an environment that expects an interface that is different from the object's interface.
  - Interface translation among multiple sources must occur.
  - An object should act as an intermediary for one of a group of classes, and it is not possible to know which class will be used until runtime.

Adapter In Code

CelciusToFahrenheitAdapter.java

```
public class CelciusToFahrenheitAdapter implements FahrenheitThermometer {

    private CelciusThermometer celciusThermometer;

    public CelciusToFahrenheitAdapter(CelciusThermometer celciusThermometer) {
        this.celciusThermometer = celciusThermometer;
    }

    public double getTemperateInF() {
        return convertCtoF(celciusThermometer.getTemperateInC());
    }

    private double convertCtoF(double c) {
        return (c * 9 / 5) + 32;
    }
}
```

Decorator Pattern

Decorator Pattern Defined

- To provide a way to flexibly add or remove component functionality without changing its external appearance or function.
- When to use:
  - You want to make dynamic changes that are transparent to users, without the restrictions of subclassing.
  - Component capabilities can be added or withdrawn as the system runs.
  - There are a number of independently varying features that you should apply dynamically, and which you can use in any combination on a component.

Decorator Pattern in Code

Sundae.java

```
public interface Sundae {
    String describeSundae();
}
```

SundaeDecorator.java

```
public abstract class SundaeDecorator implements Sundae {
    private Sundae decorated;
    public SundaeDecorator(Sundae decorated) {
        this.decorated = decorated;
    }
    public Sundae getDecorated() {
        return decorated;
    }
}
```

Specialized Decorator

FudgeSundaeDecorator.java

```
public class FudgeSundaeDecorator extends SundaeDecorator {
    public FudgeSundaeDecorator(Sundae sundae) {
        super(sundae);
    }

    public String describeSundae() {
        return "Fudge " + getDecorated().describeSundae();
    }
}
```

Facade Design Pattern

Facade Design Pattern Defined

- To provide a simplified interface to a group of subsystems or a complex subsystem.
- When to use:
  - Make complex systems easier to use by providing a simpler interface without removing the advanced options.
  - Reduce coupling between clients and subsystems.
  - Layer subsystems by providing Facades for sets of subsystems.

Select Essential Behavioral Patterns

Command Pattern

Command Pattern Defined

- To wrap a command in an object so that it can be stored, passed into methods, and returned like any other object.
- When to use:
  - Support undo, logging, and/or transactions.
  - Queue and execute commands at different times.
  - Decouple the source of the request from the object that fulfills the request.

Command Pattern in Code

CommandFactory.java

```
public class CommandFactory {
    private HashMap<String, Command> commands;

    public CommandFactory() {
        this.commands = new HashMap<String, Command>();
    }

    public void addCommand(String label, Command command) {
        this.commands.put(label, command);
    }

    public void execute(String label) {
        Command command = this.commands.get(label);
        if (command != null) {
            command.apply();
        }
    }
}
```

Observer Pattern

Observer Pattern Defined

- To provide a way for a component to flexibly broadcast messages to interested receivers.
- Has been updated with newer ideas like the `EventBus`
- When to use:
  - At least one message sender.
  - One or more message receivers that might vary within an application or among applications.

The Observable in Code

RSSFeed.java

```
public class RSSFeed {

    private List<RSSObserver> observers = new ArrayList<RSSObserver>();

    public void broadcast(RSSEntry entry) {
        for (RSSObserver observer : observers) {
            observer.update(entry);
        }
    }

    public RSSObserver addObserver(RSSObserver observer) {
        observers.add(observer);
        return observer;
    }

    public void removeObserver(RSSObserver observer) {
        observers.remove(observer);
    }
}
```

The Observer in Code

RSSObserver.java

```
public interface RSSObserver {
    void update(RSSEntry entry);
}
```

LoggingObserver.java

```
public class LoggingObserver implements RSSObserver {

    private FileWriter log;

    public LoggingObserver(File log) {
        try {
            this.log = new FileWriter(log);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void update(RSSEntry entry) {
        try {
            log.write(entry.getValue());
            log.write('\n');
            log.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Google Guice

About Google Guice

- Dependency Inject Framework
- Uses Code above Configuration
- Uses Type Safety for Component Wiring
- Applies the "Hollywood Principal", "Don’t call us we will call you"
- Tries to avoid Factories
- Billed as `@Inject` is the new `new`
- Adheres to JSR-330 (JSR-330 is the Dependency Injection Java Specification Request)

Installing Guice

Including Guice in Maven

```
<dependency>
    <groupId>com.google.inject</groupId>
    <artifactId>guice</artifactId>
    <version>4.1.0</version>
</dependency>
```

Including Guice in Gradle

```
compile 'com.google.inject:guice:4.1.0'
```

Rules about Dependency Injection

- Make the component injectable, naturally occurs with testable code
- The framework usually should not be exposed within the code.
- Other than annotations, the code should not about it's wiring be it Spring or Guice.
- Create your components based on requirements, there is a way to wire components, when those components are designed well.

Lab: Pre-Class Check

Before we begin it is assumed that all of you have the following tools installed:

- JDK 1.8.x
- Maven 3.3.x

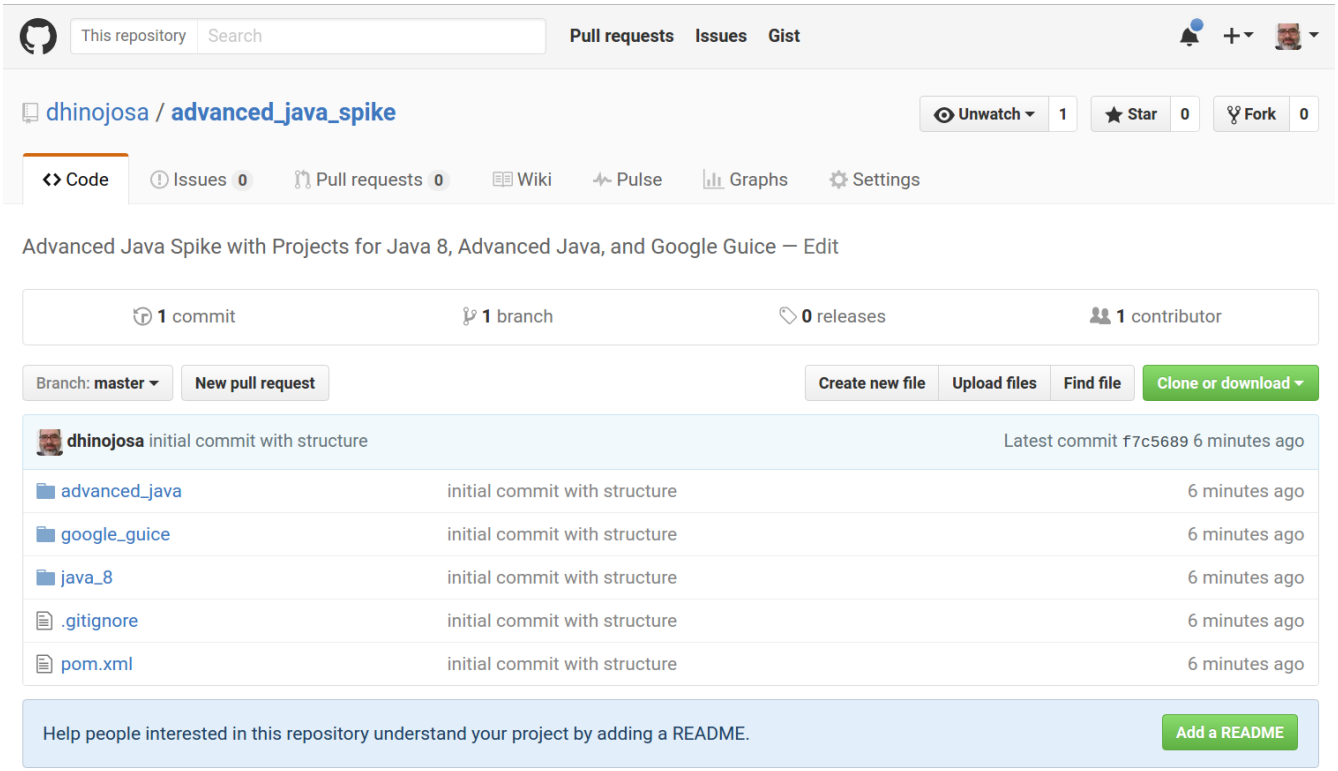
To verify that all your tools work as expected

```
% javac -version
javac 1.8.0_65

% java -version
java version "1.8.0_65"
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% mvn -v
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T09:41:47-
07:00)
Maven home: /usr/lib/mvn/apache-maven-3.3.9
Java version: 1.8.0_65, vendor: Oracle Corporation
Java home: /usr/lib/jvm/jdk1.8.0_65/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.4.0-34-generic", arch: "amd64", family: "unix"
```

NOTE The JDK 8 Version doesn't have to be exact as long as it is Java 8.

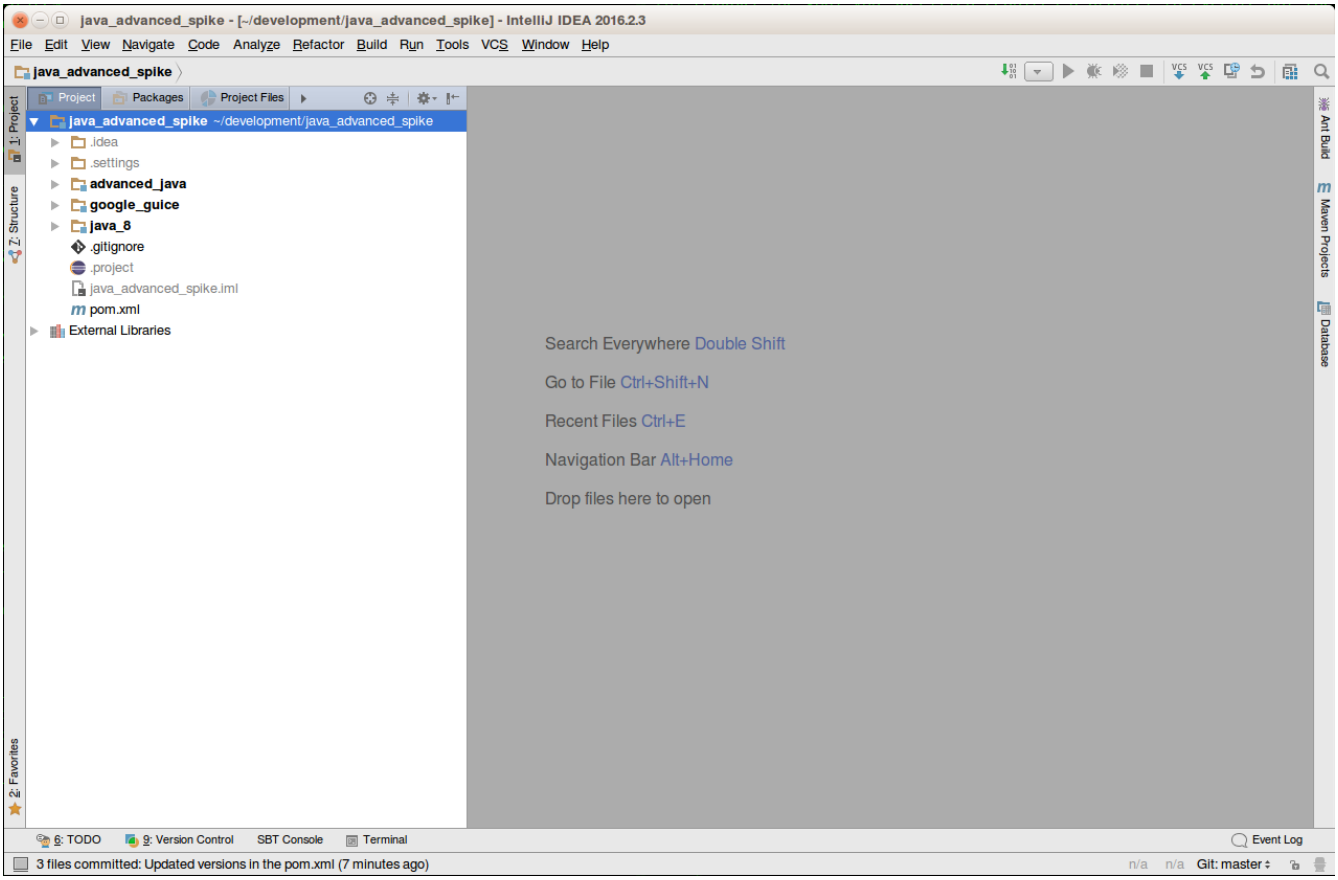
Lab: Download the three day project

From https://github.com/dhinojosa/advanced_java_spike download the project .zip file and extract it into your favorite location.



Optional Lab: Open Project in IntelliJ

Once downloaded and extracted to your favorite location, In IntelliJ Open The Project, IntelliJ will recognize it as a Maven project and you are good to go.

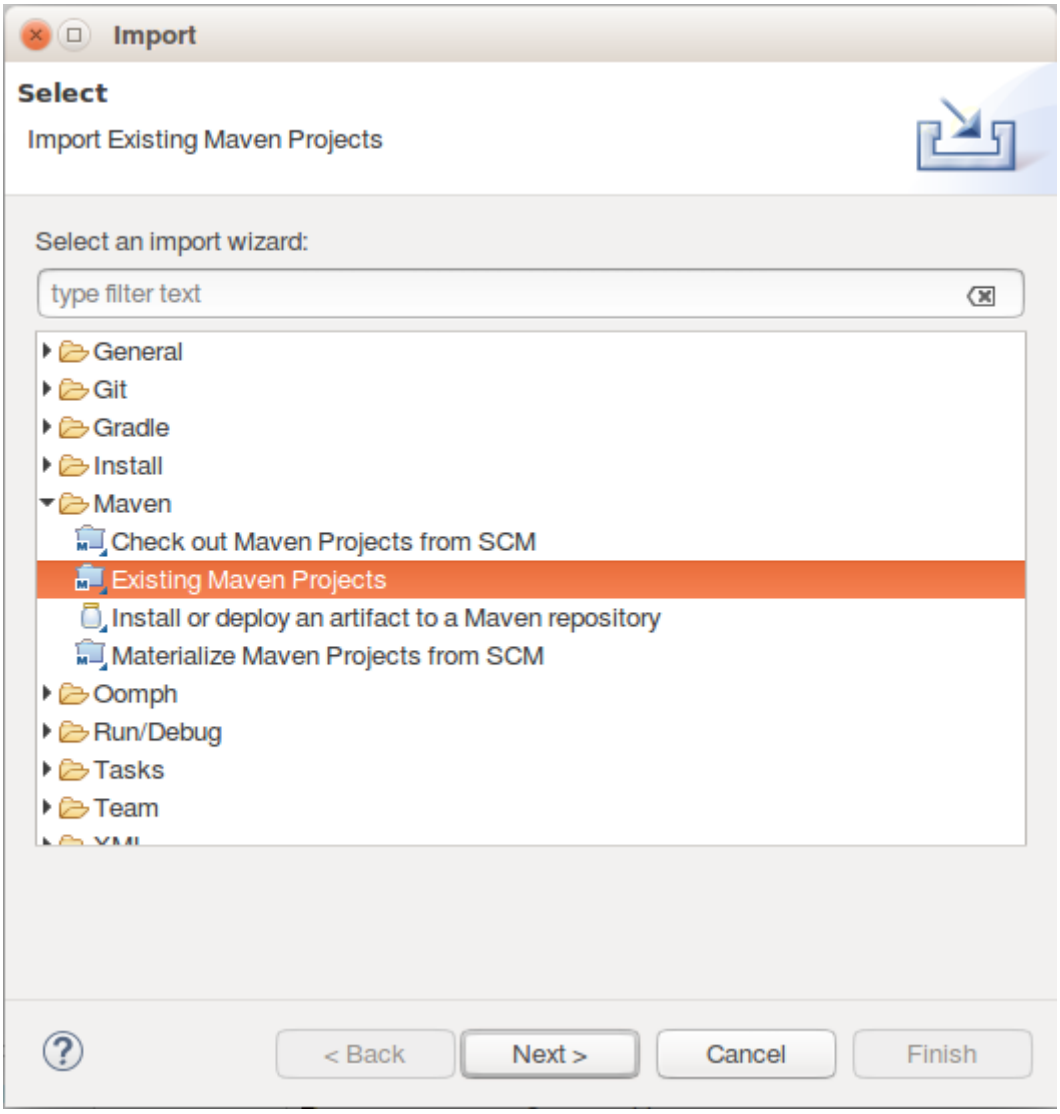


Optional Lab: Open Project in Eclipse

Once downloaded and extracted:

Step 1: Select *File* > *Import Project* in the menu.

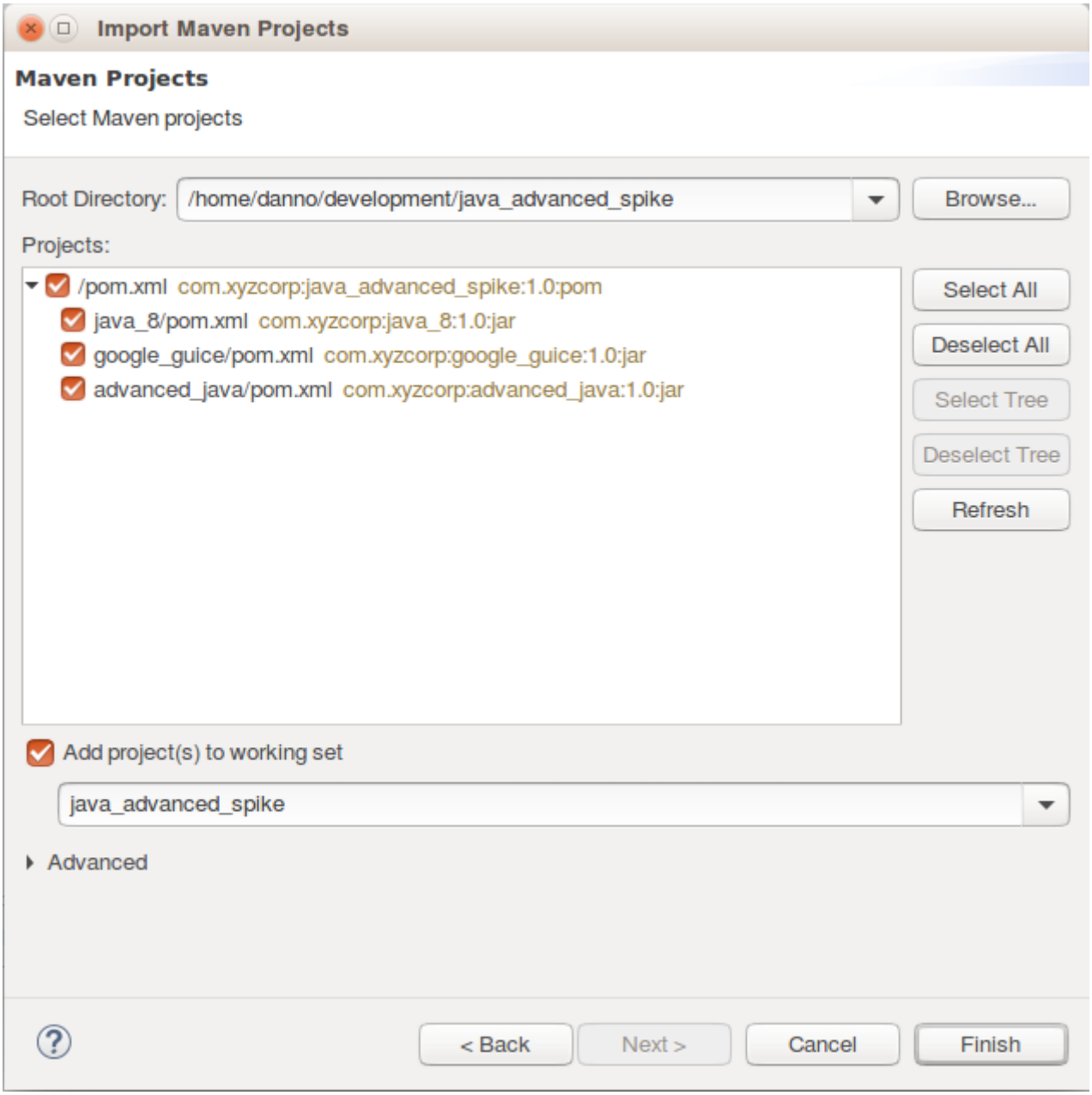
Step 2: In the following dialog box:



- Open the *Maven* category
- Select *Import Existing Maven Projects*

Optional Lab: Open Project in Eclipse (Continued)

Step 3:



- Click the *Browse:* button next to *Root Directory*
- Select the location of your ***java_advanced_spike*** directory.

Step 4: Click *Finish*

Lab: Review the Production Code and Test Code Produced.

Bootstrapping a Guice Module

- Bootstrapping of a Module is done in an implementation of an `com.google.inject.AbstractModule`
- Once extended, implement as part of the module contract `configure`
- In the configure method, use `bind` to bind components to one another.

Lab: Setting Bootstrapping a Module

Step 1: In `src/main/java` of the `google_guice` module create a package `com.xyzcorp`

Step 2: In the `com.xyzcorp` package create a java file, `AlbumModule.java`

Step 3: Create the following information.

AlbumModule.java

```
package com.xyzcorp;

import com.google.inject.AbstractModule;

public class AlbumModule extends AbstractModule {
    @Override
    protected void configure() {

    }
}
```

Linked Bindings

- Links a *type* to a concrete implementation.
- Whenever an implementation is required it can now be found using the *type*

Lab: Linked Bindings

Step 1: In the `src/main/java` directory and in the `com.xyzcorp.dao` package locate the `AlbumDAO interface`. This will be the type that will be bound.

Step 2: In the `src/main/java` directory and in the `com.xyzcorp.dao` package locate the `StandardAlbumDAO` concrete `class`. Notice that this is a concrete implementation.

Step 3: Link the abstraction to the implementation in `AlbumModule`:

AlbumModule.java

```
package com.xyzcorp;

import com.google.inject.AbstractModule;
import com.xyzcorp.dao.AlbumDAO;
import com.xyzcorp.dao.StandardAlbumDAO;

public class AlbumModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(AlbumDAO.class).to(StandardAlbumDAO.class);
    }
}
```

Lab: Create the Injector

Step 1: In the folder `src/main/java` and in the `com.xyzcorp` package create a `Runner.java` file.

Step 2: Include the following code.

Step 3: Run the code, and verify that you received a `NullPointerException` since we have more wiring to do.

```
package com.xyzcorp;

import com.google.inject.Guice;
import com.google.inject.Injector;
import com.xyzcorp.dao.AlbumDAO;

public class Runner {
    public static void main(String[] args) throws SQLException {
        Injector injector = Guice.createInjector(new AlbumModule());
        AlbumDAO dao = injector.getInstance(StandardAlbumDAO.class);
        dao.insert(new Album("Joshua Tree", "U2", 1987));
    }
}
```

Binding Annotations

- Guice uses annotation to bind multiple components for the same type.
- For example, as in our example, a Data Access Object (DAO) will have multiple bindings.
- Doing so, will require creation of annotations that can bind via `FIELD`, `PARAMETER`, and `METHOD`

Lab: Creating an H2 Annotation

Step 1: In the `src/main/java` and in the `com.xyzcorp` package, create a package called `annotations` if none are created.

Step 2: In the `com.xyzcorp.annotations` package create a file `H2.java` with the following content:

```
package com.xyzcorp.annotations;

import com.google.inject.BindingAnnotation;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@BindingAnnotation
@Target({FIELD, PARAMETER, METHOD})
@Retention(RUNTIME)
public @interface H2 {}
```

Step 3: Create `Oracle` and `MySQL` annotations for a well-rounded example.

Guice `Provider`

- As it's own limited capacity factory is the `Provider<T>`
- `Provider` is parameterized to provide what the type of what it provides.
- `Provider` are injectable in of themselves, but can be used to provide custom made objects when necessary.
- The signature of a `Provider` is the following:

Guice's Provider Signature

```
public interface Provider<T> {
  T get();
}
```

Lab: Provider Examples

Step 1: In `src/main/java` and in the package `com.xyzcorp.dao` already created for you is a `Provider` implementation.

Step 2: Review how `H2ConnectionProvider` is made.

Step 3: In `AlbumModule.java` let's map the `provider` that provides a `java.sql.Connection`

```
package com.xyzcorp;

import com.google.inject.AbstractModule;
import com.google.inject.name.Names;
import com.xyzcorp.dao.AlbumDAO;
import com.xyzcorp.dao.H2ConnectionProvider;
import com.xyzcorp.dao.StandardAlbumDAO;

import java.sql.Connection;

public class AlbumModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(Connection.class).annotatedWith(H2.class)
                              .toProvider(H2ConnectionProvider.class);
        bind(AlbumDAO.class).to(StandardAlbumDAO.class);
    }
}
```

Lab: Injecting the H2 Connection to our DAO

Step 1: In `StandardAlbumDAO`, apply the annotations `@Inject` and `@H2`.

```
@Inject
public void setConnection(@H2 Connection connection) {
   this.connection = connection;
}
```

This will inject the `java.sql.Connection` into this setter, but will inject a specific one the `H2` Connection.

Constructor Injection for Guice Provider

`@Named`

WARNING	This is not a lab just an example

- Guice comes with a built-in binding annotation `@Named` that uses a string to provide a name.
- Use this method if you wish *not* to use an annotation to discriminate what objects you wish to inject.
- For example, instead of an `@H2` annotation we could've used:

```
@Override
@Inject
public void setConnection(@Named("H2") Connection connection) {
    this.connection = connection;
}
```

- If we used the following binding in the `AlbumModule`:

```
bind(Connection.class).annotatedWith(Names.named("H2")).toProvider(H2ConnectionProvide
r.class);
```

Instance Bindings

- You can bind a type to a specific *instance* of that type using `toInstance`.
- Doing an instance is considered a Singleton, only one instance available.

Lab: Making `java.lang.String` `@Named` Instance Bindings

Step 1: In `AlbumModule`, create the following bindings for `url`, `userName`, and `password`

```
package com.xyzcorp;

import com.google.inject.AbstractModule;
import com.google.inject.name.Names;
import com.xyzcorp.annotations.H2;
import com.xyzcorp.dao.AlbumDAO;
import com.xyzcorp.dao.H2ConnectionProvider;
import com.xyzcorp.dao.StandardAlbumDAO;

import java.sql.Connection;

public class AlbumModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(String.class).annotatedWith(Names.named("url")).toInstance
                ("jdbc:h2:tcp://localhost/~/test");
        bind(String.class).annotatedWith(Names.named("userName")).toInstance
                ("sa");
        bind(String.class).annotatedWith(Names.named("password")).toInstance
                ("");
        ...
    }
}
```

Lab: Wire the named items into our provider

Step 1: The `H2ConnectionProvider` constructor requires the `url`, `userName` and `password`

Step 2: We can bring those in with the `@Named` annotation (either the `javax.inject` flavor, or the `com.google.inject.name.Named`

```
@Inject
public H2ConnectionProvider(@Named("url") String url,
                            @Named("userName") String userName,
                            @Named("password") String password) {
    this.url = url;
    this.userName = userName;
    this.password = password;
}
```

Lab: Starting an Instance of an H2 Server

Step 1: Go to your project directory

Step 2: To start an instance of an H2 Server run the following command:

Start an H2 Server on Linux/MacOSX

```
java -cp ~/.m2/repository/com/h2database/h2/1.4.192/h2-1.4.192.jar org.h2.tools.Server
```

Start an H2 Server on Windows

```
java -cp %USER_PROFILE%\.m2\repository\com\h2database\h2\1.4.192\h2-1.4.192.jar
org.h2.tools.Server
```

NOTE	You can replace `%USER_PROFILE%` with the location of your `Users` directory. e.g. `C:\Users\pedro`

Lab: Creating the Database

Step 1: Enter the following information to start the database.

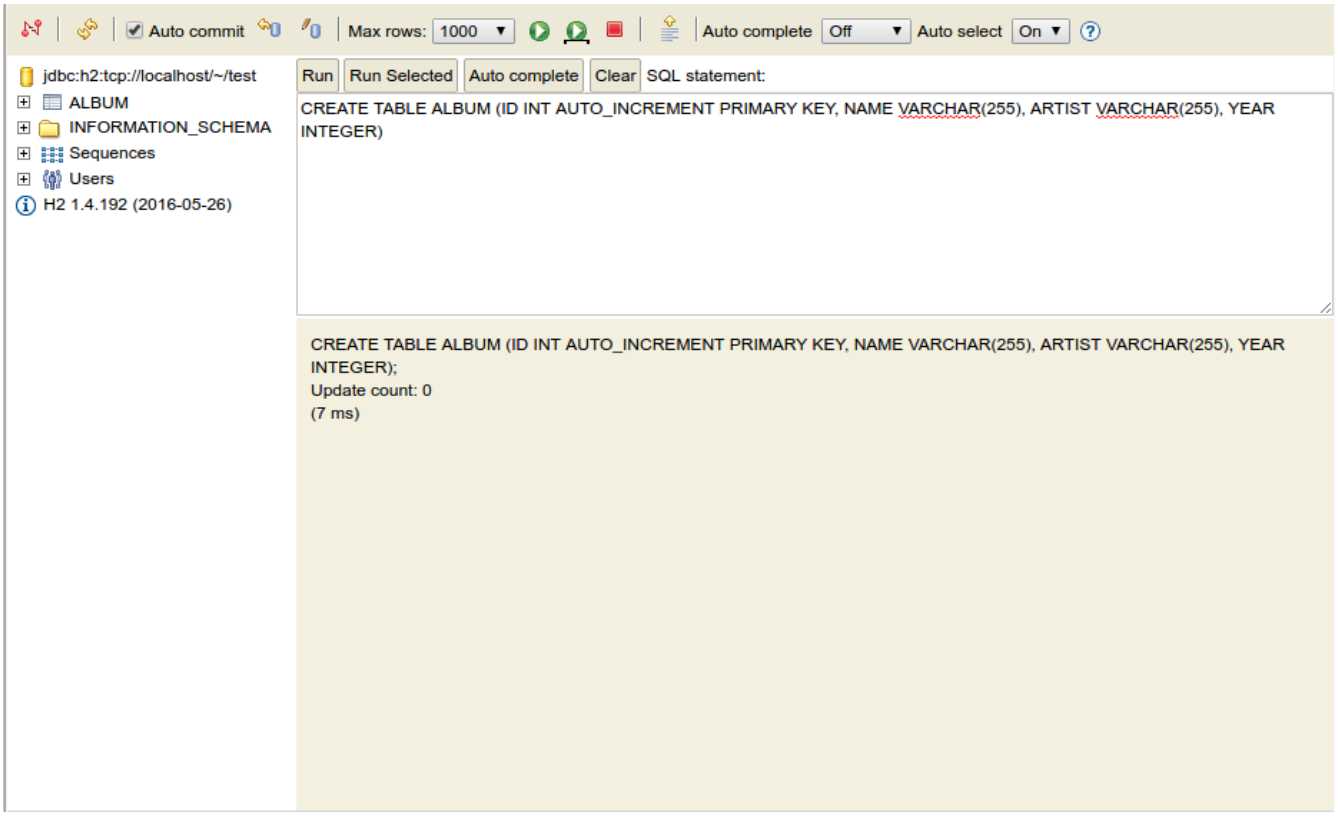
Step 2: Press 'Test Connection' to test that the connection is made.

Step 3: Press 'Connect' to Proceed to the Connection.

Lab: Create a Table in the Database

Step 1: Create an `ALBUM` Table in the Database `test` using the following SQL Statement:

```
CREATE TABLE ALBUM (ID INT AUTO_INCREMENT PRIMARY KEY, NAME VARCHAR(255), ARTIST
VARCHAR(255), YEAR INT)
```



Step 2: When done, click on `Run`

Lab: Bringing it all together!

Step 1: In `Runner`, run the `public static void main(String[] args)` in your IDE of Choice.

Step 1 (Alternate): You can also run at the command line:

```
mvn exec:java -Dexec.mainClass=com.xyzcorp.Runner
```

Step 2: Enter some of your favorite albums and enter them into your database using Guice.

Annotations @Provides

If we didn't want to actually create an actual class for a Provider we could've used the `@Provides` annotation.

With some caveats:

- The `Provider` must be declared in the `Module`
- Any `Exception` that is thrown inside of `@Provides` will in turn throw a `ProvisionException`

```
public class AlbumModule extends AbstractModule {
    @Override
    protected void configure() {
      ...
    }

    @Provides
    Connection provideConnection() {
        JdbcConnectionPool cp = JdbcConnectionPool.create(
                "jdbc:h2:tcp://localhost/~/test", "sa", "");
        try {
            return cp.getConnection();
        } catch (SQLException e) {
            throw new RuntimeException("Unable to create a connection", e);
        }
    }
}
```

Just-in Time Bindings

There are two annotations that you can use that maps the abstraction and the implementation.

- `@ProvidedBy`
- `@ImplementedBy`

These annotations do the binding without the mapping in the `AbstractModule`

`@ProvidedBy`

```
@ProvidedBy(DatabaseTransactionLogProvider.class)
public interface TransactionLog {
  void logConnectException(UnreachableException e);
  void logChargeResult(ChargeResult result);
}
```

This is the equivalent of

```
bind(TransactionLog.class)
        .toProvider(DatabaseTransactionLogProvider.class);
```

`@ImplementedBy`

```
@ImplementedBy(PayPalCreditCardProcessor.class)
public interface CreditCardProcessor {
  ChargeResult charge(String amount, CreditCard creditCard)
      throws UnreachableException;
}
```

This is the equivalent of

```
bind(CreditCardProcessor.class).to(PayPalCreditCardProcessor.class);
```

Scopes

- By Default, ***Guice returns a new instance each time it supplies a value***
- A Scope can be made by either the by the `bind` statement or annotation.
- Classes annotated `@Singleton`***must be threadsafe***`

Either annotated on the class.

```
@Singleton
public class InMemoryTransactionLog implements TransactionLog {
  /* everything here should be threadsafe! */
}
```

Bound in the `AbstractModule`

```
bind(TransactionLog.class).to(InMemoryTransactionLog.class).in(Singleton.class);
```

This can also be annotation on a `Provider<T>`

```
@Provides @Singleton
TransactionLog provideTransactionLog() {
  ...
}
```

Integration Testing Using a Module

- Consider multiple `AbstractModule`
- One will be necessary for production code.
- But the other can be used for integration testing.
- For example, the production database is wired with Guice for Production while the test

database is used for Testing and Development.

Remaining Thoughts

Thank you