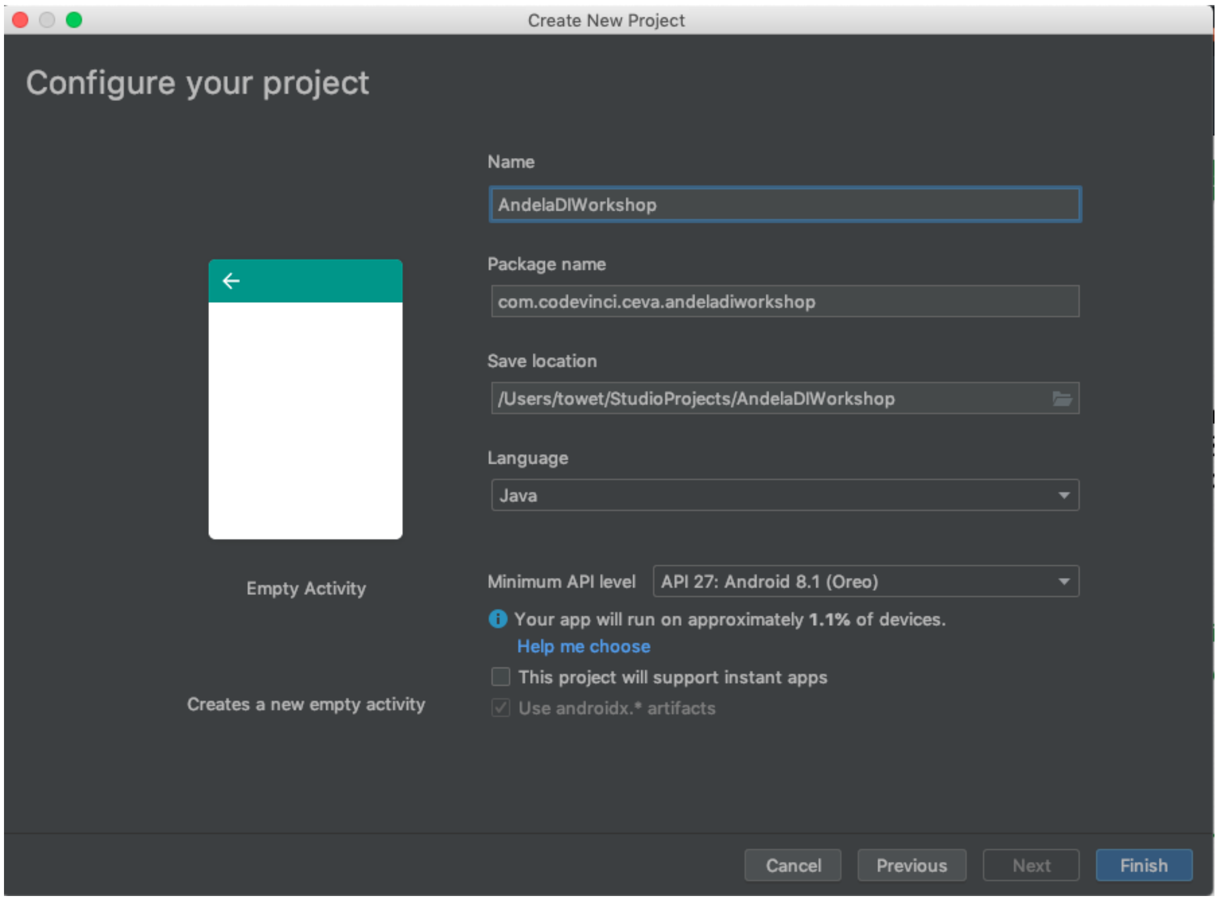
DI Code

Part 1

1. Create a new android project with empty activity



Configure your project

Name: AndelaDIWorkshop

Package name: com.codevinci.ceva.andeladiworkshop

Save location: /Users/towet/StudioProjects/AndelaDIWorkshop

Language: Java

Minimum API level: API 27: Android 8.1 (Oreo)

Your app will run on approximately **1.1%** of devices.

Help me choose

☐ This project will support instant apps

✓ Use androidx.* artifacts

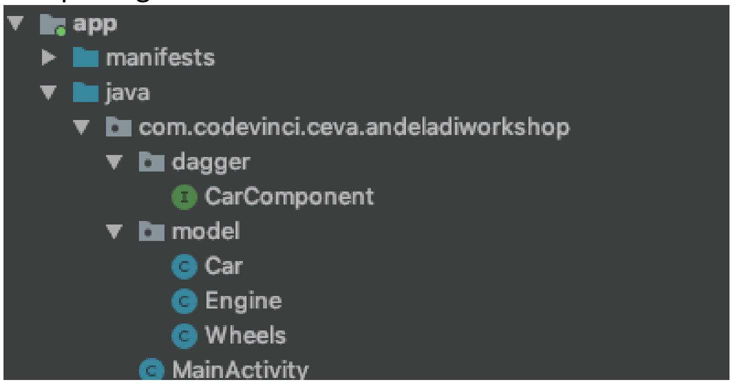
2. First we set up our gradle dependencies

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'

    //dagger
    api 'com.google.dagger:dagger:2.24'
    annotationProcessor 'com.google.dagger:dagger-compiler:2.24'
    api 'com.google.dagger:dagger-android:2.24'
    api 'com.google.dagger:dagger-android-support:2.24' // if you use the support libraries
    annotationProcessor 'com.google.dagger:dagger-android-processor:2.24'

    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

3. Our package structure will be like this



```
▼ app
    ▶ manifests
    ▼ java
        ▼ com.codevinci.ceva.andeladiworkshop
            ▼ dagger
                CarComponent
            ▼ model
                Car
                Engine
                Wheels
              MainActivity
```

4. Create an empty java class Engine.java

```java
import javax.inject.Inject;

public class Engine {

    @Inject
    public Engine(){

    }
}
```

5. Create an empty class Wheels.java

```java
public class Wheels{

    @Inject
    public Wheels(){

    }
}
```

6. Create a java class Car

```java
public class Car {
    private static final String TAG = "Car";
    private Engine engine;
    private Wheel wheel;

    @Inject //constructor injection
    public Car(Engine engine, Wheel wheel){
        this.engine = engine;
        this.wheel = wheel;
    }

    public void drive(){
        Log.d( TAG, "driving...");
    }
}
```

7. Create a java ***interface*** called CarComponent.java – this is the backbone of Dagger. This is where classes get objects they want to use.

```java
@Component //add this annotation - dagger will at compile time implement this interface
public interface CarComponent {

    Car getCar();//provision method
}
```

8. Rebuild Project
9. Paste the following in your MainActivity.java

```java
public class MainActivity extends AppCompatActivity {
    private Car car;//field injection

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /*Engine engine = new Engine();
        Wheel wheel = new Wheel();
        Car car = new Car(engine,wheel);
        car.drive();*/

        CarComponent carComponent = DaggerCarComponent.create();
        car = carComponent.getCar();
        car.drive();
    }
}
```

Part 2

What happens when you can't access the constructor of the object class? E.g when dealing with activities?

1. Update the CarComponent interface as follows

```
@Component
public interface CarComponent {
    Car getCar();

    void inject(MainActivity mainActivity);
}
```

2. Update MainActivity as follows

```
public class MainActivity extends AppCompatActivity {
    @Inject //field injection - we cant make the member private
    Car car;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        CarComponent carComponent = DaggerCarComponent.create();
        carComponent.inject(this);

        /*car = carComponent.getCar();*/
        car.drive();
    }
}
```

This process is called field injection. By calling inject we tell dagger to inject the member variables that are annotated by @Inject
Field injection is meant for framework types that the android system instantiates e.g activities and fragments

You can’t do constructor injection on activities and fragments because of the reason above.

Part 3

Special note: Field and method injection are only called after constructor injection.

1. Create a java file called Remote

```
public class Remote {
    private static final String TAG = "Car";

    @Inject
    public Remote() {
    }

    public void setListener(Car car) {
        Log.d(TAG, "Remote connected...");
    }
}
```

2. Update the Car class

```
public class Car {
    private static final String TAG = "Car";
    @Inject Engine engine; //field injection
    private Wheels wheels;

    @Inject //constructor injection
    public Car(Wheels wheels) {
        this.wheels = wheels;
    }

    @Inject //method injection - passing the injected object itself to the dependency
    public void enableRemote(Remote remote){
        remote.setListener(this);
    }
    public void drive(){
        Log.d(TAG, "driving...");
    }
}
```

Part 4

How do we interact with third party libraries or objects that we can’t annotate with @Inject

1. Create empty Java classes for Rims and Tyres
2. Update Tyres as follows

```java
public class Tyres {
    //we don't own this class so we can't annotate it with @Inject

    private static final String TAG = "Car";

    public void inflate(){
        Log.d( TAG, "tyres inflated...");
    }
}
```

3. Update Wheels

```java
public class Wheels {
    //we don't own this class so we can't annotate it with @Inject
    private Rims rims;
    private Tyres tyres;


    public Wheels(Rims rims, Tyres tyres) {
        this.rims = rims;
        this.tyres = tyres;
    }

}
```

4. Create WheelsModule.java

```java
@Module //classes that contribute to the object graph over methods
public class WheelsModule {

    //we tell dagger how it can provide rims
    @Provides
    Rims provideRims() {
        return new Rims();
    }

    @Provides
    Tyres provideTyres() {
        //we can use these provides methods to do config on objects before returning them
        Tyres tyres = new Tyres();
        tyres.inflate();
        return tyres;
    }
@Provides
    Wheels provideWheels(Rims rims, Tyres tyres) {
        return new Wheels(rims, tyres);
    }
}
```

Part 5

We will learn how and why we should use @Binds instead of @Provides to provide implementations for interfaces from our modules.

1. We convert Engine.java from a class to an interface

```
public interface Engine {
    /*@Inject
    public Engine() {
    }*/
    void start();
}
```

2. Create two classes PetrolEngine and DieselEngine that implements Engine interface
   PetrolEngine.java

```
public class PetrolEngine implements Engine {
    private static final String TAG = "Car";

    @Inject
    public PetrolEngine() {
    }

    @Override
    public void start() {
        Log.d(TAG, "Petrol engine started.");
    }
}
```

DieselEngine.java

```
public class DieselEngine implements Engine {
    private static final String TAG = "Car";
```

```java
    @Inject
    public DieselEngine() {
    }

    @Override
    public void start() {
        Log.d(TAG, "Diesel engine started.");
    }
}
```

Dagger can now instantiate these classes and provide them where needed.

3. Our car class has to provide Engine but it cant provide any randomly. What to do? We create modules.

PetrolEngineModule.java

```java
@Module
public class PetrolEngineModule {
    @Provides
    Engine provideEngine(PetrolEngine engine){
        return engine;
    }
}
```

4. Add the PetrolEngineModule to the CarComponent interface

```java
@Component(modules = {WheelsModule.class, PetrolEngineModule.class})
public interface CarComponent {
    Car getCar();

    void inject(MainActivity mainActivity);
}
```

5. Update drive() method in our Car class

```java
public void drive(){
    engine.start();//shows us which engine is set
    Log.d(TAG, "driving...");
}
```

6. We can optimize our PetrolEngineModule further by replacing @Provides with @Binds.

@Binds is more concise as we write less code. It also has better performance as Dagger does not have to instantiate the module.
You cant use @Provides and @Binds in the same module because @Provides requires that Dagger instantiates the class.

```
@Module
public abstract class PetrolEngineModule {
   @Binds
   abstract Engine bindEngine(PetrolEngine engine);
}
```

7. Create a new file – DieselEngineModule

```
@Module
public abstract class DieselEngineModule {
   @Binds
   abstract Engine bindEngine(DieselEngine engine);
}
```

8. Go to CarComponent and swap out the PetrolEngineModule for a DieselEngineModule

```
@Component(modules = {WheelsModule.class, DieselEngineModule.class})
```

See what we did there? We swapped a whole engine without even touching the Activity or Car classes. De-coupling 101

Note: We cant put multiple modules that implement the same interface – Dagger wouldn't

Part 6

We will learn how to use stateful modules to inject variables into our dependency graph at run-time. For this, we have to add a constructor to our module and manually set it on the Component Builder with the required parameters.

1. Update our DieselEngine.java

```
public class DieselEngine implements Engine {
   private static final String TAG = "Car";


   private int horsePower;

```

```
    @Inject
    public DieselEngine(int horsePower) {
        this.horsePower = horsePower;
    }

    @Override
    public void start() {
        Log.d(TAG, "Diesel engine started.");
    }
}
```

2. We have to update our DieselEngineModule back to using @Provides because it supports configuration.

```
@Module
public class DieselEngineModule {
    private int horsePower;

    public DieselEngineModule(int horsePower) {
        this.horsePower = horsePower;
    }

    @Provides
    Engine provideEngine(){
        return new DieselEngine(horsePower);
    }
}
```

3. Rebuild Project
4. Update our MainActivity

```
public class MainActivity extends AppCompatActivity {
    @Inject //field injection - we cant make the member private
    Car car;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);


        /*CarComponent carComponent = DaggerCarComponent.create();*///cant use create when we need to pass
arguments
        CarComponent carComponent = DaggerCarComponent.builder()
                .dieselEngineModule(new DieselEngineModule(150))
                .build();

        carComponent.inject(this);
        /*car = carComponent.getCar();*/
        car.drive();
    }
}
```

5. Run the project

Part 7

we will learn how to use @BindsInstance to bind variables to our dependency graph at runtime as an alternative to providing them from a stateful module. For this, we have to declare our own @Component.Builder.
To distinguish between multiple bindings from the same type, we will use the @Named qualifier.

1. Update our PetrolEngine class

```
public class PetrolEngine implements Engine {
    private static final String TAG = "Car";

    private int horsePower;

    @Inject
    public PetrolEngine(@Named("horsepower") int horsePower) {
        this.horsePower = horsePower;
    }

    @Override
    public void start() {
        Log.d(TAG, "Petrol engine started.\nHorsepower "+horsePower);
    }
}
```

2. Update CarComponent. We shall use the Builder pattern to define the API for the CarComponent builder ourselves. @BindsInstance allows us to do the builder chain calls.

```
@Component(modules = {WheelsModule.class, PetrolEngineModule.class})
public interface CarComponent {
    Car getCar();

    void inject(MainActivity mainActivity);

    @Component.Builder
    interface Builder{

        @BindsInstance
        Builder horsePower(@Named("horsepower") int horsePower);

        CarComponent build();
    }
}
```

3. Rebuild
4. Update MainActivity

```
public class MainActivity extends AppCompatActivity {
    @Inject //field injection - we cant make the member private
    Car car;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /*CarComponent carComponent = DaggerCarComponent.create();*///cant use create when we
need to pass arguments
        /*CarComponent carComponent = DaggerCarComponent.builder()
                .dieselEngineModule(new DieselEngineModule(150))
                .build();*/

        CarComponent carComponent = DaggerCarComponent.builder()
                .horsePower(150)
                .build();
        carComponent.inject(this);
        /*car = carComponent.getCar();*/
        car.drive();
    }
}
```

5. Update PetrolEngine, CarComponent and MainActivity to include a second argument – engineCapacity

PetrolEngine.java

```java
public class PetrolEngine implements Engine {
    private static final String TAG = "Car";

    private int horsePower;
    private int engineCapacity;

    @Inject
    public PetrolEngine(@Named("horsepower") int horsePower, @Named("enginecapacity") int engineCapacity) {
        this.horsePower = horsePower;
        this.engineCapacity = engineCapacity;
    }

    @Override
    public void start() {
        Log.d(TAG, "Petrol engine started.\nHorsepower: "+horsePower+"\nEngine Capacity: "+engineCapacity);
    }
}
```

CarComponent.java

```java
@Component(modules = {WheelsModule.class, PetrolEngineModule.class})
public interface CarComponent {
    Car getCar();

    void inject(MainActivity mainActivity);

    @Component.Builder
    interface Builder{

        @BindsInstance
        Builder horsePower(@Named("horsepower") int horsePower);

        @BindsInstance
        Builder engineCapacity(@Named("enginecapacity") int engineCapacity);

        CarComponent build();
    }
}
```

MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    @Inject //field injection - we cant make the member private
    Car car;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /*CarComponent carComponent = DaggerCarComponent.create();*///cant use create when we need to pass
arguments
        /*CarComponent carComponent = DaggerCarComponent.builder()
                .dieselEngineModule(new DieselEngineModule(150))
                .build();*/

        CarComponent carComponent = DaggerCarComponent.builder()
                .horsePower(150)
                .engineCapacity(2700)
                .build();
        carComponent.inject(this);
        /*car = carComponent.getCar();*/
        car.drive();
    }
}
```

@BindInstance allows us to inject variables into the dependency graph at run-time

Part 8

We will learn how to use the @Singleton scope annotation to tell Dagger to only create a single instance of an object and reuse it within the **same component.**
Internally, Dagger wraps the object's Factory into a DoubleCheck Provider, which caches the instance and uses double-checked locking to return it in a thread-safe and efficient way.

1. Create an empty Driver class

```
@Singleton
public class Driver {

   @Inject
   public Driver() {
   }
}
```

2. Annotate CarComponent with @Singleton

```
@Singleton
@Component(modules = {WheelsModule.class, PetrolEngineModule.class})
public interface CarComponent {
   Car getCar();

   void inject(MainActivity mainActivity);

   @Component.Builder
   interface Builder{

      @BindsInstance
      Builder horsePower(@HorsePower int horsePower);

      @BindsInstance
      Builder engineCapacity(@Named("enginecapacity") int engineCapacity);

      CarComponent build();
   }
}
```

3. Now update MainActivity. We shall add another instance of Car. What we want to achieve is to have one instance of driver(singleton) but two instances of car object.

```
public class MainActivity extends AppCompatActivity {
   @Inject //field injection - we cant make the member private
   Car car1, car2;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_main);

        /*CarComponent carComponent = DaggerCarComponent.create();*///cant use create when we need to pass
arguments
        /*CarComponent carComponent = DaggerCarComponent.builder()
                .dieselEngineModule(new DieselEngineModule(150))
                .build();*/

        CarComponent carComponent = DaggerCarComponent.builder()
                .horsePower(150)
                .engineCapacity(2700)
                .build();
        carComponent.inject(this);
        /*car = carComponent.getCar();*/
        car1.drive();
        car2.drive();
    }
}
```

4. Check log for object hash – you will see one object of driver.
5. Check generated code – Providers are wrapper objects that know how to create an instance of the objects they are wrapping. Providers are normally factory classes. DoubleCheck class contains the logic for the singleton
6. **Double-check locking** allows you to get thread-safe singletons in an efficient way

Part 9

We will learn how to create and use custom scopes. This is useful if we want to reuse single instances of dependencies for example in an activity or fragment, but not keep them as application-wide singletons that stay in memory for the whole lifetime of the app.
To achieve this, we have to create a custom Java Annotation which itself is annotated with **@Scope** (and the two meta-annotations **@Documented** and **@Retention**), and use it in conjunction with a Dagger Component that lives as long as this particular scope, for example in the lifecycle of an activity or as long as a user is logged in. If we want to create an application-wide singleton, we have to instantiate its component in the **Application** class and access it from everywhere else through a call to getApplication.
To connect 2 components of different scopes into one dependency graph, we can declare the outer component as a dependency of the inner component and pass it as a Builder argument. The outer component has to expose dependencies to the inner component explicitly via provision methods.

1. Create an android Application class. Call it ExampleApp

```java
public class ExampleApp extends Application {
    private CarComponent component;

    @Override
    public void onCreate() {
        super.onCreate();

        component = DaggerCarComponent.builder()
                .horsePower(400)
                .engineCapacity(4500)
                .build();
    }

    public CarComponent getAppComponent(){
        return component;
    }
}
```

2. Update android manifest

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.codevinci.ceva.andeladiworkshop">

    <application
        android:name=".ExampleApp"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

3. Create PerActivityScope file

```
@Scope
@Documented
@Retention(RUNTIME)
public @interface PerActivityScope {
}
```

4. Annotate our Car class with @PerActivityScope

```
@PerActivityScope
public class Car {
    private static final String TAG = "Car";
    private Driver driver;
    @Inject Engine engine; //field injection
    private Wheels wheels;

    @Inject //constructor injection
    public Car(Driver driver,Wheels wheels) {
        this.driver = driver;
        this.wheels = wheels;
    }

    @Inject //method injection - passing the injected object itself to the dependency
    public void enableRemote(Remote remote){
        remote.setListener(this);
    }

    public void drive(){
        engine.start();//shows us which engine is set
        Log.d(TAG, driver+ " drives "+this);
    }
}
```

5. We shall rename our CarComponent to ActivityComponent then create an AppComponent class

```
@PerActivityScope
@Component(modules = {WheelsModule.class, PetrolEngineModule.class})
public interface ActivityComponent {
    Car getCar();

    void inject(MainActivity mainActivity);

    @Component.Builder
    interface Builder{

        @BindsInstance
        Builder horsePower(@HorsePower int horsePower);

        @BindsInstance
        Builder engineCapacity(@Named("enginecapacity") int engineCapacity);

        ActivityComponent build();
    }
}
```

6. Update Driver class

```
public class Driver {
 //We cant access this class
}
```

7. Create a Driver module

```
@Module
public abstract class DriverModule {

    @Provides
    @Singleton
    static Driver provideDriver(){
        return new Driver();
    }
}
```

8. Create AppComponent class

```java
@Singleton
@Component(dependencies = ActivityComponent.class, modules = DriverModule.class)
public interface AppComponent {

    Driver getDriver();
}
```

9. Update our ExampleApp class

```java
public class ExampleApp extends Application {
    private AppComponent component;

    @Override
    public void onCreate() {
        super.onCreate();

        component = DaggerAppComponent.create();
    }

    public AppComponent getAppComponent(){
        return component;
    }
}
```

10.