BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Mobile development and security

Session 9

Karim Karimov
Lecturer

Dependency injection

Dependency injection (DI) is a technique widely used in programming and well suited to Android development. By following the principles of DI, you lay the groundwork for good app architecture.
Implementing dependency injection provides you with the following advantages:

- Reusability of code
- Ease of refactoring
- Ease of testing

Fundamentals of DI

Classes often require references to other classes. For example, a **Car** class might need a reference to an **Engine** class. These required classes are called dependencies, and in this example the Car class is dependent on having an instance of the Engine class to run.
There are three ways for a class to get an object it needs:

1. The class **constructs** the dependency it needs. In the example above, **Car** would create and initialize its own instance of **Engine**.
2. **Grab it** from somewhere else. Some Android APIs, such as Context getters and **getSystemService()**, work this way.
3. Have it supplied as a **parameter**. The app can provide these dependencies when the class is constructed or pass them in to the functions that need each dependency. In the example above, the **Car** constructor would receive **Engine** as a parameter.

The third option is **dependency injection**!

Fundamentals DI

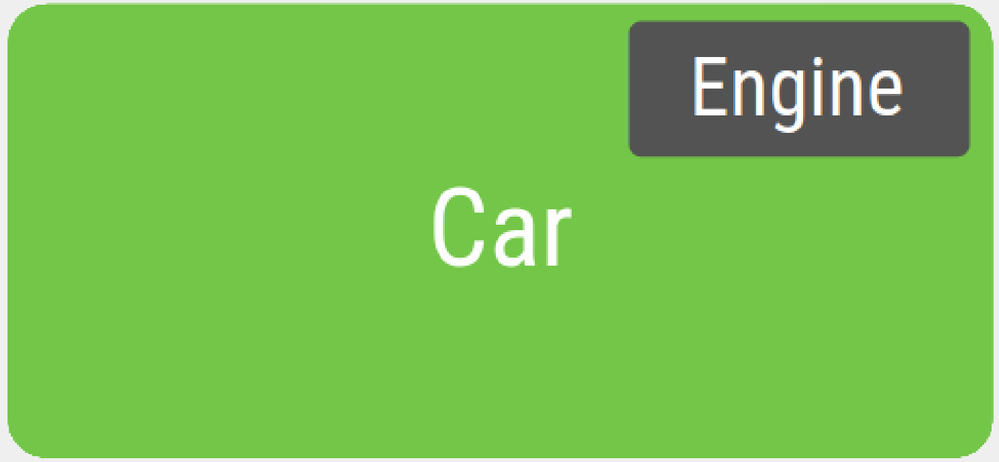
Without dependency injection, representing a Car that creates its own Engine dependency in code looks like this:

```
class Car {

    private val engine = Engine()

    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val car = Car()
    car.start()
}
```



Fundamentals DI

This is not an example of dependency injection because the Car class is constructing its own Engine. This can be problematic because:

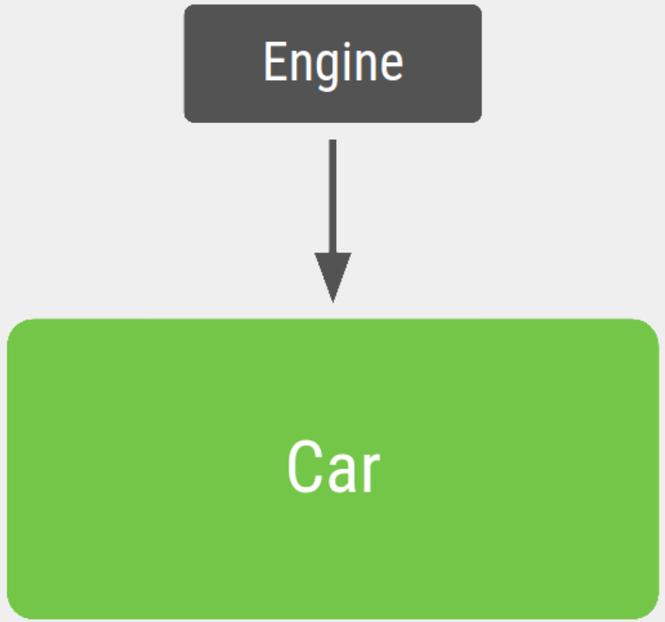
- Car and Engine are **tightly coupled** - an instance of Car uses one type of Engine, and no subclasses or alternative implementations can easily be used. If the Car were to construct its own Engine, you would have to create two types of Car instead of just reusing the same Car for engines of type ***Gas*** and ***Electric***.
- The hard dependency on Engine makes **testing more difficult**. Car uses a real instance of Engine, thus preventing you from using a test double to modify Engine for different test cases.

Fundamentals DI

What does the code look like with dependency injection? Instead of each instance of Car constructing its own Engine object on initialization, it receives an Engine object as a parameter in its constructor:

```
class Car(private val engine:
Engine) {
    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val engine = Engine()
    val car = Car(engine)
    car.start()
}
```



Fundamentals DI

The main function uses Car. Because Car depends on Engine, the app creates an instance of Engine and then uses it to construct an instance of Car. The benefits of this DI-based approach are:

- **Reusability** of Car. You can pass in different implementations of Engine to Car. For example, you might define a new subclass of Engine called **ElectricEngine** that you want Car to use. If you use DI, all you need to do is pass in an instance of the updated ElectricEngine subclass, and Car still works without any further changes.
- **Easy testing** of Car. You can pass in test doubles to test your different scenarios. For example, you might create a test double of Engine called **FakeEngine** and configure it for different tests.

Types of DI

There are two major ways to do dependency injection in Android:

- **Constructor Injection**. This is the way described on right. You pass the dependencies of a class to its constructor.

```
class Car(private val engine:
Engine) {
    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val engine = Engine()
    val car = Car(engine)
    car.start()
}
```

Types of DI

Field Injection (or Setter Injection). Certain Android framework classes such as activities and fragments are instantiated by the system, so constructor injection is not possible. With field injection, dependencies are instantiated after the class is created. The code would look like this:

```
class Car {
    lateinit var engine: Engine

    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val car = Car()
    car.engine = Engine()
    car.start()
}
```

Inversion of control

Dependency injection is based on the **Inversion of Control** principle in which generic code controls the execution of specific code.
In software engineering, inversion of control (IoC) is a design pattern in which custom-written portions of a computer program receive the flow of control from a generic framework. A software architecture with this design inverts control as compared to traditional procedural programming: in traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks, but with inversion of control, it is the framework that calls into the custom, or task-specific, code.
Inversion of control is used to increase modularity of the program and make it extensible, and has applications in object-oriented programming and other programming paradigms.

Alternative to DI

An alternative to dependency injection is using a **service locator**. The service locator design pattern also improves decoupling of classes from concrete dependencies. You create a class known as the service locator that creates and stores dependencies and then provides those dependencies on demand.

```
object ServiceLocator {
    fun getEngine(): Engine = Engine()
}

class Car {
    private val engine = ServiceLocator.getEngine()

    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val car = Car()
    car.start()
}
```

Service locator cons

The service locator pattern is different from dependency injection in the way the elements are consumed. With the service locator pattern, classes have control and ask for objects to be injected; with dependency injection, the app has control and proactively injects the required objects.
Compared to dependency injection:

- The collection of dependencies required by a service locator makes code harder to test because all the tests have to interact with the **same global service locator**.
- Dependencies are encoded in the class implementation, **not in the API surface**. As a result, it's harder to know what a class needs from the outside. As a result, changes to Car or the dependencies available in the service locator might result in runtime or test failures by causing references to fail.
- Managing lifetimes of objects is more difficult if you want to scope to anything other than the lifetime of the entire app.

DI pros

Dependency injection provides your app with the following advantages:

- **Reusability of classes and decoupling** of dependencies: It's easier to swap out implementations of a dependency. Code reuse is improved because of inversion of control, and classes no longer control how their dependencies are created, but instead work with any configuration.
- **Ease of refactoring**: The dependencies become a verifiable part of the API surface, so they can be checked at object-creation time or at compile time rather than being hidden as implementation details.
- **Ease of testing**: A class doesn't manage its dependencies, so when you're testing it, you can pass in different implementations to test all of your different cases.

Automated DI

In the previous examples, we created, **provided**, and **managed** the dependencies of the different classes yourself, without relying on a library. This is called dependency injection by hand, or **manual dependency** injection. In the **Car** example, there was only one dependency, but more dependencies and classes can make manual injection of dependencies more tedious. Manual dependency injection also presents several problems:

- For big apps, taking all the dependencies and connecting them correctly can require a large amount of **boilerplate code**. In a multi-layered architecture, in order to create an object for a top layer, you have to provide all the dependencies of the layers below it. As a concrete example, to build a real car you might need an engine, a transmission, a chassis, and other parts; and an engine in turn needs cylinders and spark plugs.
- When you're not able to construct dependencies before passing them in — for example when using lazy initializations or scoping objects to flows of your app — you **need to write and maintain a custom container** (or graph of dependencies) that manages the lifetimes of your dependencies in memory.

Automated DI

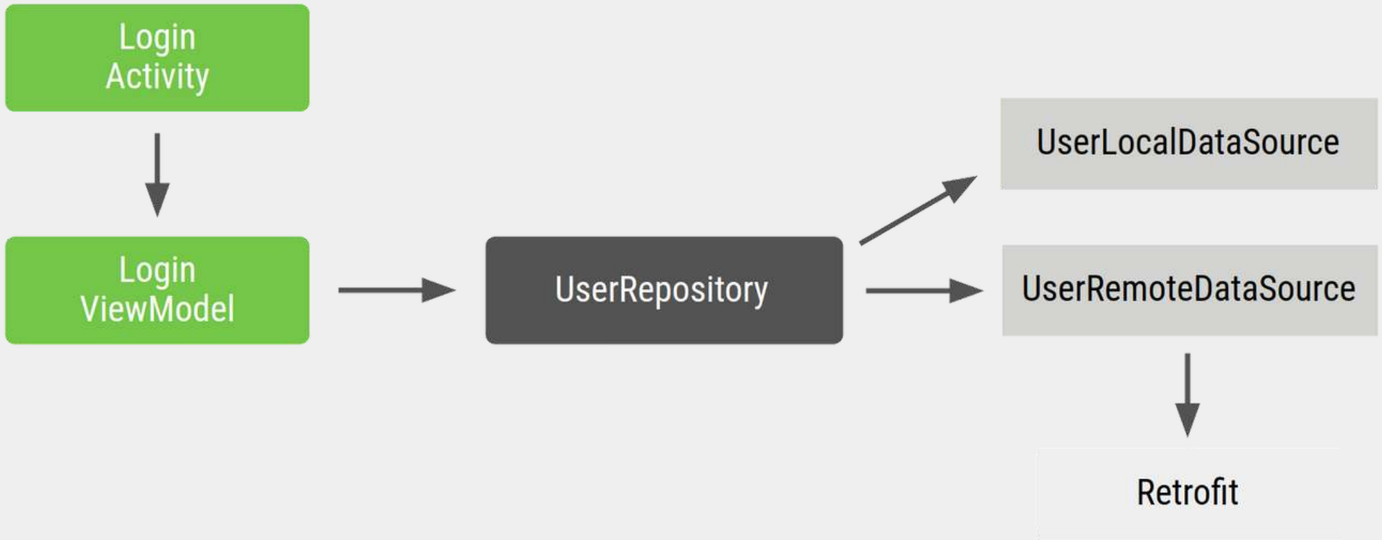
There are libraries that solve this problem by automating the process of creating and providing dependencies. They fit into two categories:

- **Reflection-based** solutions that connect dependencies at runtime.
- Static solutions that **generate the code** to connect dependencies at compile time.

Dagger is a popular dependency injection library for Java, Kotlin, and Android that is maintained by Google. Dagger facilitates using DI in your app by creating and managing the graph of dependencies for you. It provides fully static and compile-time dependencies addressing many of the development and performance issues of reflection-based solutions such as **Guice**.

Manual DI

Consider a flow to be a group of screens in your app that correspond to a feature. Login, registration, and checkout are all examples of flows.
When covering a login flow for a typical Android app, the **LoginActivity** depends on **LoginViewModel**, which in turn depends on UserRepository. Then UserRepository depends on a **UserLocalDataSource** and a **UserRemoteDataSource**, which in turn depends on a **Retrofit** service.



Manual DI

LoginActivity is the entry point to the login flow and the user interacts with the activity. Thus, LoginActivity needs to create the **LoginViewModel** with all its dependencies.
The Repository and DataSource classes of the flow look like this:

```
class UserRepository(
    private val localDataSource: UserLocalDataSource,
    private val remoteDataSource:
UserRemoteDataSource
) { ... }

class UserLocalDataSource { ... }
class UserRemoteDataSource(
    private val loginService: LoginRetrofitService
) { ... }
```

Manual DI

```
class LoginActivity: Activity() {

    private lateinit var loginViewModel: LoginViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // In order to satisfy the dependencies of LoginViewModel, you have to also
        // satisfy the dependencies of all of its dependencies recursively.
        // First, create retrofit which is the dependency of UserRemoteDataSource
        val retrofit = Retrofit.Builder()
            .baseUrl("https://example.com")
            .build()
            .create(LoginService::class.java)

        // Then, satisfy the dependencies of UserRepository
        val remoteDataSource = UserRemoteDataSource(retrofit)
        val localDataSource = UserLocalDataSource()

        // Now you can create an instance of UserRepository that LoginViewModel needs
        val userRepository = UserRepository(localDataSource, remoteDataSource)

        // Lastly, create an instance of LoginViewModel with userRepository
        loginViewModel = LoginViewModel(userRepository)
    }
}
```

Manual DI

There are issues with this approach:

1. There's a lot of **boilerplate code**. If you wanted to create another instance of LoginViewModel in another part of the code, you'd have code duplication.
2. Dependencies **have to be declared in order**. You have to instantiate UserRepository before LoginViewModel in order to create it.
3. It's **difficult to reuse** objects. If you wanted to reuse UserRepository across multiple features, you'd have to make it follow the singleton pattern. The singleton pattern makes testing more difficult because all tests share the same singleton instance.

Containers for DI

To solve the issue of reusing objects, you can create your own dependencies container class that you use to get dependencies. All instances provided by this container can be public. In the example, because you only need an instance of UserRepository, you can make its dependencies private with the option of making them public in the future if they need to be provided:

```
// Container of objects shared across the whole app
class AppContainer {

    // Since you want to expose userRepository out of the container, you need to satisfy
    // its dependencies as you did before
    private val retrofit = Retrofit.Builder()
                            .baseUrl("https://example.com")
                            .build()
                            .create(LoginService::class.java)

    private val remoteDataSource = UserRemoteDataSource(retrofit)
    private val localDataSource = UserLocalDataSource()

    // userRepository is not private; it'll be exposed
    val userRepository = UserRepository(localDataSource, remoteDataSource)
}
```

Containers for DI

```
// Custom Application class that needs to be specified
// in the AndroidManifest.xml file
class MyApplication : Application() {

   // Instance of AppContainer that will be used by all the Activities of the app
   val appContainer = AppContainer()
}
```

```
class LoginActivity: Activity() {

    private lateinit var loginViewModel: LoginViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Gets userRepository from the instance of AppContainer in Application
        val appContainer = (application as MyApplication).appContainer
        loginViewModel = LoginViewModel(appContainer.userRepository)
    }
}
```

Containers for DI

If LoginViewModel is needed in more places in the application, having a centralized place where you create instances of LoginViewModel makes sense. You can move the creation of LoginViewModel to the container and provide new objects of that type with a factory. The code for a **LoginViewModelFactory** looks like this:

```
// Definition of a Factory interface with a function to create objects of a type
interface Factory<T> {
    fun create(): T
}

// Factory for LoginViewModel.
// Since LoginViewModel depends on UserRepository, in order to create instances of
// LoginViewModel, you need an instance of UserRepository that you pass as a parameter.
class LoginViewModelFactory(private val userRepository: UserRepository) : Factory {
    override fun create(): LoginViewModel {
        return LoginViewModel(userRepository)
    }
}
```

Containers for DI

```
// AppContainer can now provide instances of LoginViewModel with LoginViewModelFactory
class AppContainer {
    ...
    val userRepository = UserRepository(localDataSource, remoteDataSource)

    val loginViewModelFactory = LoginViewModelFactory(userRepository)
}

class LoginActivity: Activity() {

    private lateinit var loginViewModel: LoginViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Gets LoginViewModelFactory from the application instance of AppContainer
        // to create a new LoginViewModel instance
        val appContainer = (application as MyApplication).appContainer
        loginViewModel = appContainer.loginViewModelFactory.create()
    }
}
```

Containers for DI

This approach is better than the previous one, but there are still some challenges to consider:

1. You have to manage AppContainer yourself, creating instances for all dependencies by hand.
2. There is still a lot of boilerplate code. You need to create factories or parameters by hand depending on whether you want to reuse an object or not.

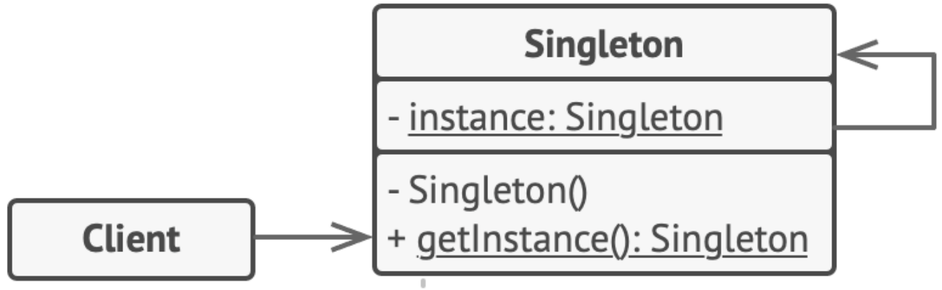
Singleton pattern

All implementations of the Singleton have these two steps in common:

- Make the **default constructor private**, to prevent other objects from using the new operator with the Singleton class.
- Create a **static creation method** that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it’s able to call the Singleton’s static method. So whenever that method is called, the same object is always returned.

Singleton pattern



1 The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

References

- **Dependency injection** - https://developer.android.com/training/dependency-injection
- **Inversion of control** - https://en.wikipedia.org/wiki/Inversion_of_control
- **Service locator** - https://en.wikipedia.org/wiki/Service_locator_pattern
- **Singleton** - https://refactoring.guru/design-patterns/singleton

End of the session