Dependency Injection

The wikipedia defination of DI is:

*“In software engineering, dependency injection is a technique in which an object receives other objects that it depends on. These other objects are called* ***dependencies”***

So far So good. matlab ki ek class kuchh kaam karna chahti hai. it needs object of some other class to complete that work and instead of creating that object by itself, it receives the object and then work on it.

eg:

**public class** Person {  
 **private** String **name**;  
 **private** Integer **age**;  
 **private** Character **gender**;  
  
 **public** Person(String name, Integer age, Character gender) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
 **this**.**gender** = gender;  
 }

}

This is a simple Person class receiving name age and gender of a person. now if I had to set up a public function that will give me all the details of it as a string , I will set up the following function inside it:

|  |  |
| --- | --- |
| **public** String getClassDetails() {  StringBuilder builder = **new** StringBuilder();  **char** endChar = **'\n'**;  builder.append(name);  builder.append(endChar);  builder.append(age);  builder.append(endChar);  builder.append(gender);  builder.append(endChar);   **return** builder.toString(); }//op:  //Ansh  //22  //M | **//RUNNER**  **public class** Tester {   **public static void** main(String[] args) {   Person p = **new** Person(**"Ansh"**,22,**'M'**);    String details = p.getClassDetails();  System.***out***.println(details);   } } |

This is all good and fine, but what if StringBuilder() class has bugs in it that we didn’t kne before? the whole function will crash and we won’t be sure weather the function is at fault or the string builder.

Under Dependecy injection pattern, the depenedies are passed to function itself

|  |  |
| --- | --- |
| **public** String getClassDetails  (StringBuilder b,**char** end){  builder.append(name);  builder.append(endChar);  builder.append(age);  builder.append(endChar);  builder.append(gender);  builder.append(endChar);  return builder.toString();  **return** builder.toString(); }  //op: Ansh 22 M | **//RUNNER**  **public class** Tester {   **public static void** main(String[] args) {   Person p = **new** Person(**"Ansh"**,22,**'M'**);    StringBuilder b = **new** StringBuilder();  **char** end = **'\t'**;    String details2 = p.getClassDetails(b,end);  System.***out***.println(details2);    } } |

We can see the obvious advantage:

since the runner is providing the dependencies , the dependencies can be easily **configured, managed and replaced**( eg, we replaced ‘\n’ by ‘\t’ for our requirement) by a central place instead of being spread throughout the application

note the 3 tasks, as they are very important:

1. **configure** : the `getClassDetails()` no longer needs to know about the configurations reqired for generating the stringbuilder .
2. **management**: the `getClassDetails()` no longer needs to ‘manage’ the instances of string builder. the runner/runner can pas the builder and automatically close it when no longer required
3. **replacement**: the dependencies can be now easily be replaced with other classes and mocks,by the caller/runner, providing a better test support

-----------------------------------------------------------------------------------------------------------------------------------------------------------------

The wiki defination goes on further saying this:

*“In software engineering, dependency injection is a technique in which an object receives other objects that it depends on. These other objects are called* ***dependencies”***.

In the typical "using" relationship the receiving object is called a **client** and the passed (that is, "injected") object is called a **service.** The code that passes the service to the client can be many kinds of things and is called the **injector**.

And

**Instead of the client specifying which service it will use, the injector tells the client what service to use.** The "injection" refers to the passing of a dependency (a service) into the object (a client) that would use it.

The second line is also somewhat clear: in our stringbuilder example,

* the runner function, aka main() would be known as **injector**
* the Person class’s getClassDetails() would be known as **client**
* the StringBuilder Object would be known as **service**.

***client***(getClassDetails*())* ***uses a service***(StringBuilder obj*)* ***“injected” via the Injector***(main())

But in the 3rd line of definition, things get interesting. According to the principle of DI, the client won’t specify which service to use. instead, it will specify which features it wants to use and thus expects the service to have.

Thus, if we have to follow this principle in our example, it would look something like this:

|  |  |
| --- | --- |
| **client(person.class)** | Injector(Tester.class’s main function) |
| **public** String getClassDetails  (X builder, **char** endChar) {  builder.append(**name**);  builder.append(endChar);  builder.append(**age**);  builder.append(endChar);  builder.append(**gender**);  builder.append(endChar);  **return** builder.toString(); } | **public class** Tester {  **public static void** main(String[] args) {  StringBuilder b = **new** StringBuilder();  **char** end = **'\t'**;  String details2 = p.getClassDetails(b,end);  System.***out***.println(details2);   **CatBuilder** c = **new** CatBuilder();  **char** end2 = **','**;  String details3 = p.getClassDetails(c,end2);  System.***out***.println(details3);    } |

Here cat builder is a class that has all the functions(append(integer), append(char), append(String),and toString() ) written inside it based on client’s requirements to use those functions. the client does not know weather its getting an instance of string builder or catbuilder, it just knows that such the object its receiving has all th function it requires for the task to be done.

But but... how could this be done in code? How can 2 classes be replacables? I think you know this. **INTERFACES(or sometimes child classe I guess?)**!!

Interfces play a major role in di and thus are the 4th important component of DI.

Recalling Interfaces and extended classes

**Interfaces** act as rules to enforce certain methods and variables for different classes. If 2 classes Ram nd Ravi implements interface Gender{ public char getGender() } , then both of these classes have to add a function getGender() in themselves, and to every other function, class , variable ,interface, etc in the project, the 2 classes will be identical in an aspect that they both have a getGender() function

The abstract classes, generic classes and the inheritance also could be used in similar maner, but that’s for some other time.

|  |  |
| --- | --- |
| **//enforcement interface**  **public interface** GenderEnforcement {  **public boolean** getGender(); } | **//main client who no longer cares weather it receives the RAM or RAVI class instance**  **public static void** checkGender(GenderEnforcement g){  System.***out***.println(g.getGender()); } |
| **// classes with getGender function enforced by the interface**  **public static class** Ravi **implements** GenderEnforcement {  **@Override public boolean** getGender() { **return false**; } }  **public static class** Ram **implements** GenderEnforcement {  **@Override public boolean** getGender() { **return false**; } } | **//main runner**  **public static void** main(String[] args) {  *checkGender*(**new** Ram());  *checkGender*(**new** Ravi()); } |

Thus we now have 4 important parties involved in DI:

* The **client** , who wants to use particular features of a service(Person class’s getClassDetails())
* The **injector**, who is willing to provide a service with required features to the client(runner function, aka main())
* The **interface**, which defines a standad set of features based on client’s requirement(Some interface X)
* The **service** itself , which is replaceable and non unique and could be anything that implements the set of features defined by the interface.(StringBuilder, CatBuilder, etc)

And 3 important problems / solved by DI :

1. **anomalous configurations of service** : the client can try to configure the service by itself as per its needs, which would make difficult to detect issues at the time of code failure.The client should not be configuring the service on its own, the service should rather be preconfigured as per client’s requirement. *DI gives a pre configured service to client*
2. **non centralised/duplication of services** : if the client is creating service instances on its own at different places, it is both a resource intensive operation and code repetition and thus a testing headache. *DI will provide the service to the client, this means the service is both centralised and created only once(or multiple times but only if required), and thus could be tested easily and there will be less duplication*
3. **tight coupling with the service :** if the client is declaring a dependency itself, its working is completely dependent on the working of the dependency itself. if the dependency fails, the client itself fails. It is not seperable with the dependency therby causing difficulties in testing. *DI make client to be independent of service and thus less coupled with the service itself*
4. Following the sequence in service declaration: ?? I don’t get it

Note: in the second point, one could argue, that if we make a service singleton, we are not repeating the code anymore. that’s correct, but singletons are difficult to test because they share the same instance

|  |
| --- |
| **A timeline of android patterns and testing tehniques**   * 2014 : fat activities(all code inside activity) was a norm. Robolectric was used as a framework for testing, which was basically like clone of android apis but instead of running on devices, it ran on jvm .   + Disadvantage : It adopted behaviors of every new android sdk release therefoe companies has to wait for new Roboelectric version release. * 2015 : MVP/MVVM stated gaining mass acceptence. activities were now thinner, only holding the views and display logic most of the time, while keeping the business logic and models separate. Junit tests were covering more and more parts of the app   + Dis advantage There was no dagger, so it was still a problem to test some methods that depended on statics, singletons and dependencies * 2016-2019/20/.. The dagger/koin/kodein/hilt/toothpick.. |

|  |
| --- |
| Dependency Injection and Service Locator : (also , dagger vs koin vs kodein)  damn,... a lot of things there. in summary, service locator is anoher approach towards one of the DI’s guiding principles : Inversion of Control (which is a whole lot of theory in itself). from what I get in abrief reading, DI is “injecting the services into client”. If we further go into it , there is this concept of injector or di container, but idk.  Service locator is , from what I understand , “client trying to use a service provided by a 3rd class called service locator”, which is somewhat similar to injector or DI container. but here , the client is itself responsible for choosing whichever sevice it wants.  finally people say SL is an anti pattern . also, people say koin is a Service locator, but is still easy to use for their projects. JW says its for the kids xD . ok bye  resources:    image I draw for di vs servicelocator(right side) in regards to main and person’s get details eg  <https://www.csharpstar.com/inversion-of-control-dependency-injection-and-service-locator-in-csharp/> : this article has good images(but aweful examples  <https://en.wikipedia.org/wiki/Service_locator_pattern>  This article calls service locator an anti pattern, but you still gotta learn koin/kodein if you wanna thrive in community. <https://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/>  <https://www.reddit.com/r/androiddev/comments/8ch4cg/dagger2_vs_koin_for_dependency_injection/>  this so has very nice answers : <https://stackoverflow.com/questions/1557781/whats-the-difference-between-the-dependency-injection-and-service-locator-patte?answertab=votes#tab-top> |

|  |
| --- |
| singletons are bad (Idk why I wandered here)  <https://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons>  [https://dzone.com/articles/singleton-anti-pattern #:~:text=The%20use%20of%20Singletons%20makes,Singleton%20and%20other%20is%20not](https://dzone.com/articles/singleton-anti-pattern%20%20%20%20%20%20%20%20#:~:text=The%20use%20of%20Singletons%20makes,Singleton%20and%20other%20is%20not). |

|  |
| --- |
| builder vs factory vs singleton  via Singleton pattern:  - constructor of class A is private.  - only an internal instance of A is available throughout the running of A and that too via  A's static method.  - So basically whenever we need A's instance, we are getting a blackbox configured  (i.e unknown how its configured inside the A class), univerally shared instance. (i.e only 1  instance is ever created)  - eg Room db  via builder pattern:  - A's constructor could be private or public. so A's instance might be manually creatable  - Some class B known as builder is generally used to create A's instance.  - Its interesting to be used because B will create a minimum dependency instance of A,  accessible via some build() method But it is upto the caller to add more dependencies  as per their requirement.  - thus multiple instances of A could be created wih different configurations, via B.build()  - eg. Retrofit client  via factory instance:  - A's constructor is public. and would require a lot of dependencies,some of which might  not be accessible to user.  - Some class F known as factory will already have all of the dependencies required to create  A. when user initialise an instance of F, all of A's dependencies will get created.  When A calls F.create(), all the dependecies will be supplied to a new instance of A  - thus multiple instances of A could be created wih SAME configuration, via F.create()  - eg viewmodel |

Manual Dependency injection, A practical example.

covered in project .../dagger/manual\_di. basically we imagined a practical mvvm based app which would use networking/viewmodels/repository etc to work.

Note: Building the app is not required since its just a prototype

===DAGGER AND HILT============================

Since google seems to be focusing on hilt now, even placing it before dagger in their documentation, I think we should start with hilt.

doubts as of now:

how to use overlaoded functions (same name different params) for creating different attachments in field injection

in any injection, what if our service could be made with different configuration in terms of service(like in one constructor we can allow user to pass an argument, but in another, we simply make a 0 arg constructor and use default values)?