

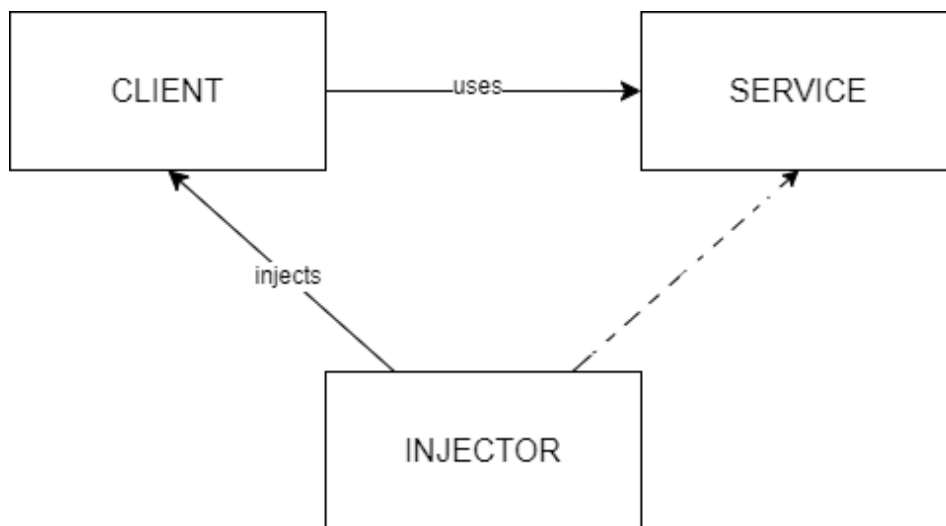
DEPENDENCY INJECTION

Introduction:

In general, when we want to develop a new application, we don't simply want to obtain working code, but we want to do it in the most effective, efficient, modular and understandable way possible. To this end, over the years, a series of rules and best practices to follow have been created, which are called "design patterns".

Mark Seemann has defined dependency injection as "a set of software design principles and patterns that enable us to develop loosely coupled code". The idea behind this principle is that we don't want our classes and modules to be dependent on other components by directly instantiate them, but we would like to have them "injected" by someone else. In this way, we can create non hard-coded dependencies that lead to more flexible and maintainable code.

There are three types of dependency injection that differ from the point in which they apply the injection, but, in general, there are always the following elements:



- Client: the class that needs the dependency.
- Service: the class that is needed. It is often the implementation of a more general interface to satisfy the dependency inversion principle. In this way, the client won't have an explicit dependency on the service class.
- Injector: the class that creates the service and injects it into the client. It can be implemented in many languages thanks to the mechanisms they offer, such as reflection and annotations. Another solutions are ready-to-use frameworks and containers.

There are a lot of tools and frameworks for every programming language that support dependency injection, such as Spring for Java and Angular for Javascript.

Constructor injection:

This type of injection uses the constructor to pass in the dependencies of a service. In this way, we are saying that, if you want to use the client, you need to provide certain classes before. Otherwise, you won't be able to use it.

```
public class MyClass {  
  
    private Dependency dependency;  
  
    public MyClass (Dependency dep){  
        this.dependency = dep;  
    }  
}
```

The class that needs the dependency must expose a public constructor that takes an instance of the required dependency as a constructor argument. A better solution can also prevent null parameters by throwing an exception (guard pattern):

```
public class MyClass {  
  
    private Dependency dependency;  
  
    public MyClass (Dependency dep) throws Exception {  
        if (dep == null) {  
            throw new Exception("cannot pass a null parameter!");  
        } else {  
            this.dependency = dep;  
        }  
    }  
}
```

Constructor injection should be used every time the client requires the service in order to work properly and when the dependency in question has a lifetime longer than a single method.

Property injection (or setter injection):

In this case, dependencies are set using setter methods:

```
public class MyClass {  
  
    private Dependency dependency;  
  
    public void setDependency(Dependency dep) throws Exception {  
        if (dep == null) {  
            throw new Exception("cannot pass a null argument!");  
        } else {  
            this.dependency = dep;  
        }  
    }  
}
```

The class that needs the dependency must expose a public setter method that takes an instance of the required dependency as an argument. This approach allows for flexibility in setting dependencies after the client class instance has been created. Property injection is mostly used when we want to use different implementations of the service during the client lifecycle or when the dependency is optional. In these cases, we usually instantiate a default service implementation for the client during its creation. At a later stage, we can switch the dependency implementation by using the setter method or decide to maintain the default configuration.

```
public class MyClass {
    private Dependency dependency;

    public MyClass(){
        this.dependency = new DefaultDependency();
    }

    public void setDependency(Dependency dep) throws Exception {
        if (dep == null) {
            throw new Exception("cannot pass a null argument!");
        } else {
            this.dependency = dep;
        }
    }
}
```

```
public class App {
    public static void main(){
        MyClass c = new MyClass();
        /*
         * Using MyClass with default dependency
         */
        try {
            c.setDependency(new SpecializedDependency());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        /*
         * Using MyClass with new dependency injected
         */
    }
}
```

Method injection:

Similarly to property injection, dependencies are injected via methods, but they can be different from setters.

```

public class MyClass {

    public void myMethod(Dependency dep) throws Exception {
        if (dep == null) {
            throw new Exception("cannot pass a null argument!");
        } else {
            /*
             * Using dependency...
             */
        }
    }
}

```

Method injection is best used when the dependency can vary with each method call. This can be the case when the **DEPENDENCY** itself represents a value, but is often seen when the caller wishes to provide the consumer with information about the context in which the operation is being invoked. This is often the case in add-in scenarios where an add-in is provided with information about the runtime context via a method parameter. In such cases, the add-in is required to implement an interface that defines the injecting method(s).

Implementing dependency injection:

Dependency injection works by using reflection or annotations to inspect the components and their dependencies, and then using dynamic proxies or code generation to create and inject the dependencies at runtime. Implementation of dependency injection can be done through an injector that works in three steps:

- 1) Scan: the injector scans the classpath for classes decorated with predefined annotations that indicate the necessity of dependency injection
- 2) Wire: the injector inspects constructor and methods for annotations that indicate the points where to execute the injection of the matching dependencies from a component registry.
- 3) Init: the injector inspects injected components for annotations that indicate methods that must be executed before and/or after their initialization or destruction (these methods are called “lifecycle callbacks”).

A practical example:

- 1) Define an annotation for marking the components that need dependency injection:

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Injectable {
}

```

- 2) Define an annotation for marking the methods and constructors that need dependency injection:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CONSTRUCTOR)
public @interface Inject {
}
```

- 3) Define the injector class (full code at <https://github.com/FabioGentili99/project-work-on-Distributed-Systems-M>):

```
public class Injector {

    private final Map<Class<?>, Object> instances;

    public Injector() {
        instances = new HashMap<>();
    }

    public void register(Class<?> type, Object instance) {
        instances.put(type, instance);
    }

    public Object inject(Class<?> targetClass) throws IllegalAccessException, InvocationTargetException, InstantiationException {
        /*
         * scan, wire and init
         */
    }
}
```

It contains a map that stores the instances of the dependencies to be used for injection (it could have been a configuration file from which to read). The inject method allows to inject the dependencies into a target object by scanning constructors that present `@Inject` annotation and assigning instances that correspond to their parameter types.

- 4) Annotate the classes and the constructors that need dependency injection with annotation defined above:

```
@Injectable
public class Client {
    private Service service;

    public Client(){
        this.service = null;
    }

    @Inject
    public Client(Service service){
        this.service = (Service) service;
    }
}
```

- 5) Create an instance of the injector and register the dependencies. At the end, we can retrieve an instance of our client with all dependencies injected:

```

public class Main {
    public static void main(String[] args) {
        Injector injector = new Injector();
        injector.register(Service.class, new MyService());

        Client client;
        try {
            client = (Client) injector.inject(Client.class);
        } catch (IllegalAccessException | InstantiationException | InvocationTargetException e) {
            throw new RuntimeException(e);
        }
        client.start();
    }
}

```

Benefits:

It may seem that dependency injection only adds complexity to the application, without improving the performance and the quality of the code. In reality, it permits us to achieve an important objective: loose coupling between client and service implementation. This leads to many advantages that can be resumed in:

- Maintainability: simple and stand-alone classes are easier to fix than tightly coupled classes. Given that maintenance costs often exceed the cost of building the code, anything that can improve maintainability is a good thing.
- Testability: loosely coupled classes that only do one thing are more prone to unit testing. Furthermore, if we pass dependencies to classes, it's quite simple to pass-in a test double implementation. If dependencies are hard-coded, it's impossible to create test doubles and we will need to modify the code.
- Readability: code that uses dependency injection is more straightforward. Constructors aren't cluttered and filled with logic, classes are clearly defined, smaller and more compact.
- Flexibility: loose coupled code can adapt to changes in requirements or in the execution environment.
- Extensibility: by relying on abstractions instead of implementations, code can be extended with different implementations of the service interface.
- Scalability: Dependency injection simplifies the process of scaling applications by promoting a modular and loosely coupled architecture. New components can be added or existing ones modified without impacting the overall system, making it easier to accommodate changing requirements and scale the application as needed.