Dependency Injection Principle

This principle goes well with Dependency inversion.

Dependencies

- a Java class has a dependency on another class, if it uses an instance of this class
  - referred to as a class dependency
  - a class which accesses a logger service has a dependency on this service class
- java classes should be as independent as possible from other Java classes
  - increases the possibility of reusing these classes and to be able to test them independently from other classes
- if a Java class creates an instance of another class via the new operator, it cannot be used (and tested) independently from this class
  - this is called a hard dependency
- dependency injection solves these “hard” dependencies

Problems with **Hard Dependency**

- this kind of dependency is very trivial in programming
  - however, when the application’s code gets bigger and more complex, the hard-coded dependency among classes introduces many problems
- the code is inflexible
  - hard to maintain and extend as when a class permanently depends on another class
  - change to the depending class my require change to the dependent class
- the code is hard to get under unit test
  - when you want to test only the functionalities of a class, you have to test other depending classes as well
- the code is hard for reuse because the classes are tightly coupled
- dependency injection solves these drawbacks by making the code more flexible to changes, easy for unit testing and reusable

Dependency Injection

- dependency injection is a technique whereby one object supplies the dependencies of another object
  - enables you to replace dependencies without changing the class that uses them
- a dependency is an object that can be used (a service)
- an injection is the passing of a dependency to a dependent object (a client) that would use it
- allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable
- dependency injection is one form of the broader technique of dependency inversion
  - supports the dependency inversion principle
- the client delegates the responsibility of providing its dependencies to external code (the injector)

Reminder (Dependency Inversion principle)

- you can introduce interfaces to break the dependencies between higher and lower level classes
  - both classes depend on the interface and no longer on each other
  - this is the central focus of the dependency inversion principle
- you still have a dependency on the lower level class with the dependency inversion principle
  - interface only decouples the usage of the lower level class but not its instantiation
  - you still need to instantiate the implementation of the interface
- goal of the dependency injection technique is to remove the above dependency by separating the usage from the creation of the object
  - reduces the amount of required boilerplate code and improves flexibility

4 roles in dependency injection

- if you want to use dependency injection, you need classes that fulfill four basic roles
  - the service you want to use
  - the client that uses the service
  - an interface that is used by the client and implemented by the service
  - the injector which creates a service instance and injects it into the client
- you already implement three of these four roles by following the dependency inversion principle
  - the service and the client are the two classes between which the dependency inversion principle intends to remove the dependency by introducing an interface
- the injector is the only role that is not required by the dependency inversion principle

Injection Types

- there are at least three ways an object can receive a reference to an external object

- constructor injection
  - the dependencies are provided through a class constructor

- setter injection
  - the client exposes a setter method that the injector uses to inject the dependency

- interface injection
  - the dependency provides an injector method that will inject the dependency into any client passed to it
  - clients must implement an interface that exposes a setter method that accepts the dependency

Constructor injection

- this method requires the client to provide a parameter in a constructor for the dependency

```
// Constructor
Client(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

Setter injection

- this method requires the client to provide a setter method for the dependency

```
// Setter method
public void setService(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

Interface injection

- this is simply the client publishing a role interface to the setter methods of the client's dependencies
  - can be used to establish how the injector should talk to the client when injecting dependencies

```
// Service setter interface.
public interface ServiceSetter {
    public void setService(Service service);
}

// Client class
public class Client implements ServiceSetter {
    // Internal reference to the service used by this client.
    private Service service;

    // Set the service that this client is to use.
    @Override
    public void setService(Service service) {
        this.service = service;
    }
}
```

```
public class Service
{
        void inject(Client c) {
                c.setService(this);
        }
}
```