Dependency Injection Explained - The easy way

By: Ryan Desmond

Dependency Injection, Inversion of Control, IoC containers, what it's all about?

For all the confusion surrounding this topic, and the expansive IoC containers supporting it (ie, Spring), it's actually pretty simple. In this short article I'm going to see if I can't explain it in the simplest possible terms.

At it's core, Dependency Injection is about decoupling a class from it's dependencies. So what is a dependency? Generally speaking, if your class (ie: DatabaseConnection) depends on another class (ie: MySQLDatabaseConnection) to do what it needs to do then you have yourself a dependency issue. In this case MySQLDatabaseConnection is a dependency of DatabaseConnection.

```
1  public class DBAccess {
2
3      public void insert(String sql){
4
5          // the line below creates a strict dependency
6          MySQLDatabaseConnection mysql = new MySQLDatabaseConnection();
7
8          mysql.insert(sql);
9
10     }
11 }
```

At first glance, this could potentially work great. But what happens when you need to insert something into an Oracle or AWS RDS instance? In the example above you'll need to fundamentally change the DatabaseConnection class to accommodate the expanded functionality.

So what's the solution? Dependency Injection. There are two fundamental types of Dependency Injection. Constructor Injection and Setter Injection. Let's take a look at Constructor Injection first. With constructor injection you pass in the dependency as argument to the constructor. To make the most of Dependency Injection your constructor should actually take in an interface as the method parameter. By doing this, your constructor can now accept any class that implements said interface and set it as (in this case) the DatabaseConnection dependency (which is now an instance variable) when the object is created.

```
public class DBAccess {

    // Instance variable DatabaseConnection is an interface
    DatabaseConnection db;

    // Constructor takes an interface as an argument - you pass a DatabaseConnection implementation
    DBAccess(DatabaseConnection db){
        this.db = db;
    }

    public void insert(String sql){

        // Now we use our inserted dependency
        db.insert(sql);

    }
}
```

There you have it. No confusing or overwhelming IoC containers required. We have just leveraged Dependency Injection to decouple our DBAccess class from its (formerly) MySQLDatabaseConnection dependency. With the above example we can now pass any class that implements DatabaseConnection to our DBAccess class.

Now let's take a look at Setter Injection. Setter Injection is very similar to Constructor Injection but the Injection happens in the setter method rather than the constructor. You can actually leverage both in a single class. In this case, that would be useful if you want to use the same object to quickly write to multiple databases. Let's take a look.

```
public class DBAccess {

    // Instance variable DatabaseConnection is an interface
    DatabaseConnection db;

    // Here we use the setter method to inject the DatabaseConnection dependency
    // This setter must be called by the calling class before the insert() can be called
    public void setDatabaseConnection(DatabaseConnection db){
        this.db = db;
    }

    public void insert(String sql){
        // Now we use our inserted dependency
        db.insert(sql);
    }
}
```

Here you can see that we are now passing the DatabaseConnection to the setter method. As noted in the comments, you'll need to call this setter method and pass it the DatabaseConnection you want before being able to call insert(). It's your call as to whether you'd like to use Constructor Injection, Setter Injection or both.

```
public class DBAccess {

    // Instance variable DatabaseConnection is an interface
    DatabaseConnection db;

    // Constructor Injection
    DBAccess(DatabaseConnection db){
        this.db = db;
    }

    // Setter Injection
    public void setDatabaseConnection(DatabaseConnection db){
        this.db = db;
    }

    public void insert(String sql){
        // Now we use our inserted dependency
        db.insert(sql);
    }
}
```

That really it. That's Dependency Injection. No IoC container required. Now that we have that covered, let's take a look at how Spring can help you with Dependency Injection (DI). Spring, and other IoC containers add a third method of Dependency Injection called Field Injection. With Field Injection, using Spring all you have to do is add the @Autowired annotation to the dependency class variable. Spring will then create the DatabaseConnection bean and inject it whenever and wherever it is appropriate, such as our class below.

```
public class DBAccess {

    // Below we use Spring's @Autowired annotation to inject the dependency at the field level
    @Autowired
    DatabaseConnection db;


    public void insert(String sql){
        // Now we use our inserted dependency
        db.insert(sql);
    }
}
```

One note, in order for Spring to be able to manage the DatabaseConnection implementation bean, any class that you want to be automatically injected using the @Autowired annotation must be annotated with @Service. In this case, for example, the MySQLDatabaseConnection (which implements the DatabaseConnection interface) would need to be annotated with @Service.

```java
@Service
public class MySQLDatabaseConnection implements DatabaseConnection{

    @Override
    public void insert(String sql){
        //...
    }

}
```

Note: the @Service annotation is also required on your dependency beans when using Constructor and Setter Injection if you are using Spring's IoC container to manage your dependencies.

Field Injection is the most convenient form of DI but it also has some drawbacks. The largest of which is that by using Field Injection you have introduced a strong dependency on the IoC container that will be responsible for injecting those dependencies. Thereby negating, in effect, the idea of decoupling your class from it's dependencies. I found this article (http://vojtechruzicka.com/field-dependency-injection-considered-harmful/) by Vojtech Ruzicka to be very helpful in articulating why a developer might reconsider using Field Injection.

There you have it. Dependency Injection is pretty simple after all. In order to decouple your classes from their dependencies you simply pass your dependencies to their required classes using constructor, setter, or field injection. In the case of constructor and setter injection no IoC container is required. To get the most out of DI, have your constructors and setters take in an interface as the dependency, then you can pass in any class that implements said dependency.

Happy coding!