Dependency Injection or Inversion of Control

Dependency Injection

> *Dependency Injection generally means passing a dependent object as a parameter to a method, rather than having the method create the dependent object.*

What it means in practice is that the method does not have a direct dependency on a particular implementation; any implementation that meets the requirements can be passed as a parameter.

How it Works

Pass in Dependencies

Here we have a normal class. The only difference is that we allow the dependencies to be passed in.

```
public class RealBillingService implements BillingService {
  private final CreditCardProcessor processor;
  private final TransactionLog transactionLog;

  public RealBillingService(CreditCardProcessor processor, TransactionLog transactionLog) {
    this.processor = processor;
    this.transactionLog = transactionLog;
  }

  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
     } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

Testing

Because our dependencies are passed in via the constructor, we can pass in mock implementations for testing.

```
public class RealBillingServiceTest extends TestCase {

  private final PizzaOrder order = new PizzaOrder(100);
  private final CreditCard creditCard = new CreditCard("1234", 11, 2010);

  private final InMemoryTransactionLog transactionLog = new InMemoryTransactionLog();
  private final FakeCreditCardProcessor processor = new FakeCreditCardProcessor();

  public void testSuccessfulCharge() {
    RealBillingService billingService
        = new RealBillingService(processor, transactionLog);
    Receipt receipt = billingService.chargeOrder(order, creditCard);

    assertTrue(receipt.hasSuccessfulCharge());
    assertEquals(100, receipt.getAmountOfCharge());
    assertEquals(creditCard, processor.getCardOfOnlyCharge());
    assertEquals(100, processor.getAmountOfOnlyCharge());
    assertTrue(transactionLog.wasSuccessLogged());
  }
}
```

To use our class, we need to create the dependencies and pass them to the constructor.

```
public static void main(String[] args) {
    CreditCardProcessor processor = new PaypalCreditCardProcessor();
    TransactionLog transactionLog = new DatabaseTransactionLog();
    BillingService billingService
        = new RealBillingService(processor, transactionLog);
    ...
  }
```

Dependency Injection Frameworks

Two popular dependency injection frameworks are Spring and Google Guice.

Using Guice

The dependency injection pattern leads to code that's modular and testable, and Guice makes it easy to write. To use Guice in our billing example, we first need to tell it how to map our interfaces to their implementations. This configuration is done in a Guice module, which is any Java class that implements the Module interface:

```
public class BillingModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(TransactionLog.class).to(DatabaseTransactionLog.class);
    bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
    bind(BillingService.class).to(RealBillingService.class);
  }
}
```

We add @Inject to RealBillingService's constructor, which directs Guice to use it. Guice will inspect the annotated constructor, and lookup values for each parameter.

```
public class RealBillingService implements BillingService {
  private final CreditCardProcessor processor;
  private final TransactionLog transactionLog;

  @Inject
  public RealBillingService(CreditCardProcessor processor,TransactionLog transactionLog) {
    this.processor = processor;
    this.transactionLog = transactionLog;
  }

  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
     } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

Finally, we can put it all together. The Injector can be used to get an instance of any of the bound classes.

```
public static void main(String[] args) {
    Injector injector = Guice.createInjector(new BillingModule());
    BillingService billingService = injector.getInstance(BillingService.class);
    ...
  }
```

Using Spring

The injection in Spring is either done via setter injection of via construction injection. These classes which are managed by Spring must conform to the JavaBean standard. In the context of Spring classes are also referred to as beans or as Spring beans.

The Spring core container:

- handles the configuration, generally based on annotations or on an XML file (XMLBeanFactory)
- manages the selected Java classes via the BeanFactory

The core container uses the so-called bean factory to create new objects. New objects are generally created as Singletons if not specified differently.

The injection in Spring is either done via setter, field or constructor injection. Classes which are managed by Spring DI must conform to the Java bean standard.

In the context of Spring classes are also referred to as beans or as spring beans.

Example

Define interface

```
package com.company.model;

public interface IWriter {
    void writer(String s);
}
```

Create some implementations

```
package com.company.model;

public class Writer implements IWriter {
    @Override
    public void writer (String s){
        System.out.println(s);
    }
}
```

```
package com.company.model;

public class NiceWriter implements IWriter {
    @Override
    public void writer (String s){
        System.out.println("The string is " + s);
    }
}
```

```
package com.company.model;

public class MySpringBeanWithDependency {
    private IWriter writer;

    public void setWriter(IWriter writer) {
        this.writer = writer;
    }

    public void run() {
        String s = "This is my test";
        writer.writer(s);
    }
}
```

The class MySpringBeanWithDependency contains a setter for the actual writer.

We will use the Spring Framework to inject the correct writer into this class.

Using Annotations

Add the **@Service** annotation the MySpringBeanWithDependency and NiceWriter.

Also define with **@Autowired** on the setWriter method that the property "writer" will be autowired by Spring.

@Autowired will tell Spring to search for a Spring bean which implements the required interface and place it automatically into the setter.

```
package testbean;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import writer.IWriter;

@Service
public class MySpringBeanWithDependency {
    private IWriter writer;

    @Autowired
    public void setWriter(IWriter writer) {
        this.writer = writer;
    }

    public void run() {
        String s = "This is my test";
        writer.writer(s);
    }
}
```

```
package writer;

import org.springframework.stereotype.Service;

@Service
public class NiceWriter implements IWriter {
    public void writer(String s) {
        System.out.println("The string is " + s);
    }
}
```

Create the following configuration class:

```
package com.company.model;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = { "com.company.model" })
public class Config {
}
```

Change your application to "only" trigger dependency injection instead of starting a full Spring Boot application.

```
package com.company.model;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;


public class Application {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
        MySpringBeanWithDependency springBean = context.getBean(MySpringBeanWithDependency.class);
        springBean.run();
        context.close();
    }
}
```

References

Reference	URL
Dependency Injection & Inversion of Control	https://www.youtube.com/watch?v=EPv9-cHEmQw
Using dependency injection in Java - Introduction - Tutorial	https://www.vogella.com/tutorials/DependencyInjection/article.html
Guice - Motivation	https://github.com/google/guice/wiki/Motivation
Dependency Injection with the Spring Framework - Tutorial	https://www.vogella.com/tutorials/SpringDependencyInjection/article.html