DEPENDS ON

SpringTutors January 10, 2016 Features 71 Views

Depends On

If a class uses the functionality of of other class inside it, then those classes are directly dependent on each other. But every class may not directly dependent on other, sometimes there may be indirect dependency. Let us consider a use case to understand.

We have a class LoanCalculator, it has a method calIntrest to it we pass principle, year, city name as input. When we call this method it should find the applicable rate of interest for the city we passsed by fetching it from properties file.

```
package com.don.bean;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
public class LoanCalculator {
private Properties props;
public void calIntrest(int principle,int year,String city) throws IOException
{
 InputStream is=new FileInputStream(new File("E:/New folder/EclipseApps/City.properties.txt"));
 props=new Properties();
 props.load(is);
//giving the city name it will give rate of interest for that city
 String roi=(String) props.get(city);
 float roIntrest=Float.parseFloat(roi);
 System.out.println("Rate of Intrest for"+city+"  = "+roIntrest+"%");
 float totalIntrest=(principle*year*roIntrest)/100;
 System.out.println("Total Intrest = "+totalIntrest);
 float totalAmount=principle+totalIntrest;
 System.out.println("Total Amount to be paid after 5 years = "+totalAmount);
}
}
```

```
package com.don.bean;
import java.io.IOException;
public class CalculatorTest {
public static void main(String[] args) throws IOException {
 LoanCalculator lc=new LoanCalculator();
 lc.calIntrest(200000, 5,"Hyderabad");
 }
}
```

The problem with the above application is we will have to read the same data repeatedly for every call to the calInterest(…) method of the LoanCalculator class.

So instead we want to cache the data so that the LoanCalculator now can go and get the data from cache. So for this when we call calInterest(…) method we will check whether the data is available with cache or not. If it is not available we will read the city and rate of interest values from properties file as key and

values and place it in properties collection. Now we will store this properties collection into the cache as shown below-

```
package com.don.bean;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
public class LoanCalculator {
 private Properties props;
 public void calIntrest(int Principle,int year,String city) throws IOException
 {
 Cache c=Cache.getInstance();
 if(c.containsKey("CityRI")==false)
 {
 System.out.println("CityRI is not avilable");
 InputStream is=new FileInputStream(new File("E:/New folder/EclipseApps/City.properties.txt"));
 props=new Properties();
 props.load(is);
 c.put("CityRI",props);
String roi=(String) props.get(city);
 System.out.println("Rate Of Intrest : "+roi+"%");
 }
 else{
 String roi=(String) props.get(city);
 System.out.println("Rate Of Intrest : "+roi+"%");
}
}
 public void roiDemo(String city)
 {
 System.out.println("I M getting the file from Cache");
 String roi=(String) props.get(city);
 System.out.println("Rate Of Intrest : "+roi+"%");
 }
}
```

Now the problem is here the city and rate of interest values may not only required for LoanCalculator there may be several other classes in the application want to read the data from the Cache. So, we need to write the logic for checking the data is available in Cache or not if not load the data into Cache and use it.

So, we end up in duplicating the same logic across various places in the application. Instead we can write the logic for populating the data into Cache in Cache constructor, because constructor of the Cache will be called only once as it is singleton and we need to read the data only once, so the better place to write the logic for populating is in constructor of the class as shown bellow-

Cache

```
package com.don.bean;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ConcurrentHashMap;
public class Cache {
private static Cache instance;
private Map<String,Object> mapdata;
private Properties props;
private Cache()
{
mapdata=new ConcurrentHashMap<String, Object>();
```

```
//write the logic for reading the data from properties file
InputStream is=new FileInputStream(new File("E:/New folder/EclipseApps/City.properties.txt"));
props=new Properties();
//put that file into properties collection
props.load(is);
//store collection into datamap
mapdata.put("CityRI",props);
}
 public static synchronized Cache getInstance() {
 if(instance==null)
 {
 instance=new Cache();
 }
 return instance;
 }
 public boolean containsKey(String key) {
 return mapdata.containsKey(key);
}
 public Object get(String key)
 {
 return mapdata.get(key);
 }
public void put(String CityRI, Properties props) {
 mapdata.put(CityRI,props);
 }
}
```

now the problem with the above code is Cache is exposed to the details of the source from where the data is coming from. Due to which any changes to underlying source again will affect the cache.

Instead we should never write the logic for populating the data into cache rather we should write that logic in separate class called CacheManager. CacheManager is one which is responsible for reading the data from underlying source system and massaging the data and storing the data into Cache.

CacheManager

```
package com.don.bean;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
public class CacheManager {
private Cache cache;
private Map<String,Object> mapData;
private static CacheManager instance;
private Properties props;
private Object object;
private CacheManager() throws FileNotFoundException, IOException
{
 mapData=new HashMap<String, Object>();
 cache=Cache.getinstance();
//write the logic for reading the data from properties file
InputStream is=new FileInputStream(new File("E:/New folder/EclipseApps/City.properties.txt"));
props=new Properties();
//put that file into properties collection
props.load(is);
//store collection into datamap
mapdata.put("CityRI",props);
 cache.put("CityRI",mapData);
}
public static synchronized CacheManager getInstance() throws FileNotFoundException, IOException
{
 if(instance==null)
 {
 instance=new CacheManager();
```

```
 }
 return instance;
}
}
```

Again the problem with the above code is the data comes from several source systems again we will write multiple access related logic in one single class. This may become complicated and dfificult to manage and modify.

Instead write the logic for reading the data from source system in Accessor class. Now CacheManager will talk to Accessor in retrieving the data and populating into cache.

```
package com.don.bean;
import java.io.FileNotFoundException;
import java.io.IOException;
public interface IAccessor {
Object getData() throws FileNotFoundException, IOException;
}
```

```
package com.don.bean;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
public class Accessor implements IAccessor {
private Properties props;
 @Override
 public Object getData() throws IOException
 {
 InputStream is=new FileInputStream(new File("E:/New folder/EclipseApps/City.properties.txt"));
 props=new Properties();
 props.load(is);
 return props;
 }
}
```

```
package com.don.bean;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
public class CacheManager {
private Cache cache;
private IAccessor accessor;
private Map<String,Object> mapData;
private static CacheManager instance;
private Object object;
private CacheManager() throws FileNotFoundException, IOException
{
 mapData=new HashMap<String, Object>();
 accessor=new Accessor();
 object=accessor.getData();
 cache=Cache.getinstance();
 cache.put("CityRI",object);

}
public static synchronized CacheManager getInstance() throws FileNotFoundException, IOException
{
 if(instance==null)
 {
 instance=new CacheManager();
 }
 return instance;
```

```
}
}
```

Now if we observe LoanCalculator is dependent on Cache or CacheManager to read the data. LoanCalculator never talks to CacheManager rather it reads the data from Cache. But inorder to populate the data into Cache, the CacheManager should be created first than Loancalculator.

So here the LoanCalculator and cacheManager are indirectly dependent on each other. Before the LoanCalculator gets created, always the CacheManager has to be created first. This is called creational dependencies these can be managed using depends on as shown below-

```
package com.don.accessor;

import java.io.IOException;
public interface IAccessor {

Object getData() throws IOException;

}
```

```
package com.don.accessor;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
public class CityRateOfIntrestPropertiesAccessor implements IAccessor {
private Properties props;
 public Object getData() throws IOException {
 props=new Properties();
 InputStream is=new FileInputStream(new File("E:/New folder/EclipseApps/City.properties.txt"));
 props.load(is);
 return props;
 }
}
```

```
package com.don.util;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import com.don.accessor.IAccessor;
public class CacheManager {
private IAccessor accessor;
private Cache cache;
private Map<String,Object> mapData;
public CacheManager(Cache cache,IAccessor accessor) throws IOException
{

 this.cache=cache
this.accessor=accessor;
 mapData=new HashMap<String, Object>();
 load();
}
public void load() throws IOException
{
 Object data=accessor.getData();
 cache.put("CityRI",data);
}
}
```

```
package com.don.util;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
public class Cache {
```

```
private Map<String,Object> mData;
public Cache()
{
 mData=new ConcurrentHashMap<String,Object>();
}
 public void put(String key, Object data) {
 mData.put(key,data);
 }
 public Object get(String key)
 {
 return mData.get(key);
 }
 public boolean containsKey(String key)
 {
 return mData.containsKey(key);
 }
}
```

```
package com.don.bean;
import java.util.Properties;
import com.don.util.Cache;
public class LoanCalculator {
private Cache cache;
private Object obj;
Properties props;
public void calIntrest(long principle,int tenure,String city) throws Exception
{
 obj=cache.get("CityRI");
if(obj==null)
 {
 throw new Exception("Cache is not initialized");
 }
 if(cache.containsKey("CityRI")==false)
 {
 throw new Exception("Invalid Key");
 }
props=(Properties)obj;
String roi=props.getProperty(city);
float rate_of_intrest=Float.parseFloat(roi);
System.out.println("Rate Of Intrest for "+city+"= "+rate_of_intrest);
float totalIntrest=(principle*tenure*rate_of_intrest)/100;
System.out.println("Total Intrest = "+totalIntrest);
float totalAmount=principle+totalIntrest;
System.out.println("Total amount to be paid after "+tenure+" year = "+totalAmount);
}
public LoanCalculator(Cache cache) {
 super();
 this.cache = cache;
}
}
```

Spring Bean Configuration File:-

application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
<bean id="cache" class="com.don.util.Cache"/>
<bean id="cacheManager" class="com.don.util.CacheManager">
<constructor-arg ref="cache"/>
<constructor-arg ref="accessor"/>
</bean>
<bean id="accessor" class="com.don.accessor.CityRateOfIntrestPropertiesAccessor"/>
<bean id="loanCalculator" class="com.don.bean.LoanCalculator" depends-on="cacheManager">
<constructor-arg ref="cache"/>
</bean>
```

```
</beans>
```

```
Properties File
City.properties.txt
```

```
Hyderabad=12.2
Banglore=13.4
Mumbai=14.2
Delhi=13.7
```

```
package com.don.test;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.don.bean.LoanCalculator;
```

```
public class DONTest {
 public static void main(String[] args) throws Exception {
 BeanFactory factory=new XmlBeanFactory(new ClassPathResource("com/don/common/application-context.xml"));
 LoanCalculator lc=(LoanCalculator)factory.getBean("loanCalculator");
 lc.calIntrest(20000,5,"Hyderabad");
}
}
```