Worksheet: Refactoring a Hello World application using Dependency Injection

based on a presentation by Sang Shin ex-Sun Java Evangelist

This worksheet explores the ideas of dependency injection and the Spring framework by exploring step-by-step refactoring processes for a standard `Hello World` application.

Lab Exercises

1. Import all the sample code for this lab
2. Build and Run `HelloWorld` sample application
3. Build and Run `HelloWorldWithCommandLineArguments` sample application
4. Build and Run `HelloWorldDecoupled` sample application
5. Build and Run `HelloWorldDecoupledInterface` sample application
6. Build and Run `HelloWorldDecoupledInterfaceWithFactory` sample application
7. Build and Run `HelloWorldSpring` sample application
8. Build and Run `HelloWorldSpringWithDI` sample application
9. Build and Run `HelloWorldSpringWithDIXMLFile` sample application
10. Build and Run `HelloWorldSpringWithDIXMLFileConstructorArgument`
11. Build and Run `HelloWorldSpringWithAnnotation` sample application
12. Build and Run `HelloWorldSpringWithAutoscan` sample application

Some of the images are taken from an Eclipse project but the equivalent can be easily achieved in IntelliJ or Netbeans.

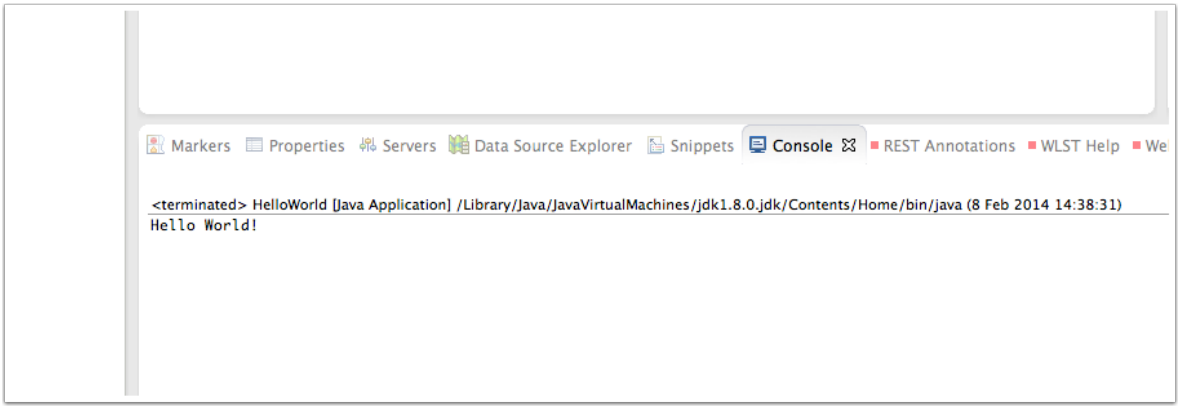
1 Download the source code repository

1. Download the repository which includes the examples for this worksheet. The relevant code will be under the folders `di` and `springdi`.
2. Create a new project called `helloworld` (or whatever name of your choice). As there are several versions of the code it is worth spending time organising the code into sub-projects (modules).

2 Build and Run `HelloWorld` sample application

In this exercise, you are going to build the good old `HelloWorld` application you probably wrote as your first Java application many years ago...

1. As most IDEs perform *incremental compilation* there is no need for a separate compilation step.
2. Run the application.
3. Observe the result.



4. Open and study `HelloWorld.java`.
   - Expand `HelloWorld->src` (or whatever the project is called).
   - Expand `helloworld`
   - Double-click `HelloWorld.java`.

HelloWorld.java

```
package helloworld;

public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

3 Build and Run `HelloWorldWithCommandLineArguments` sample application

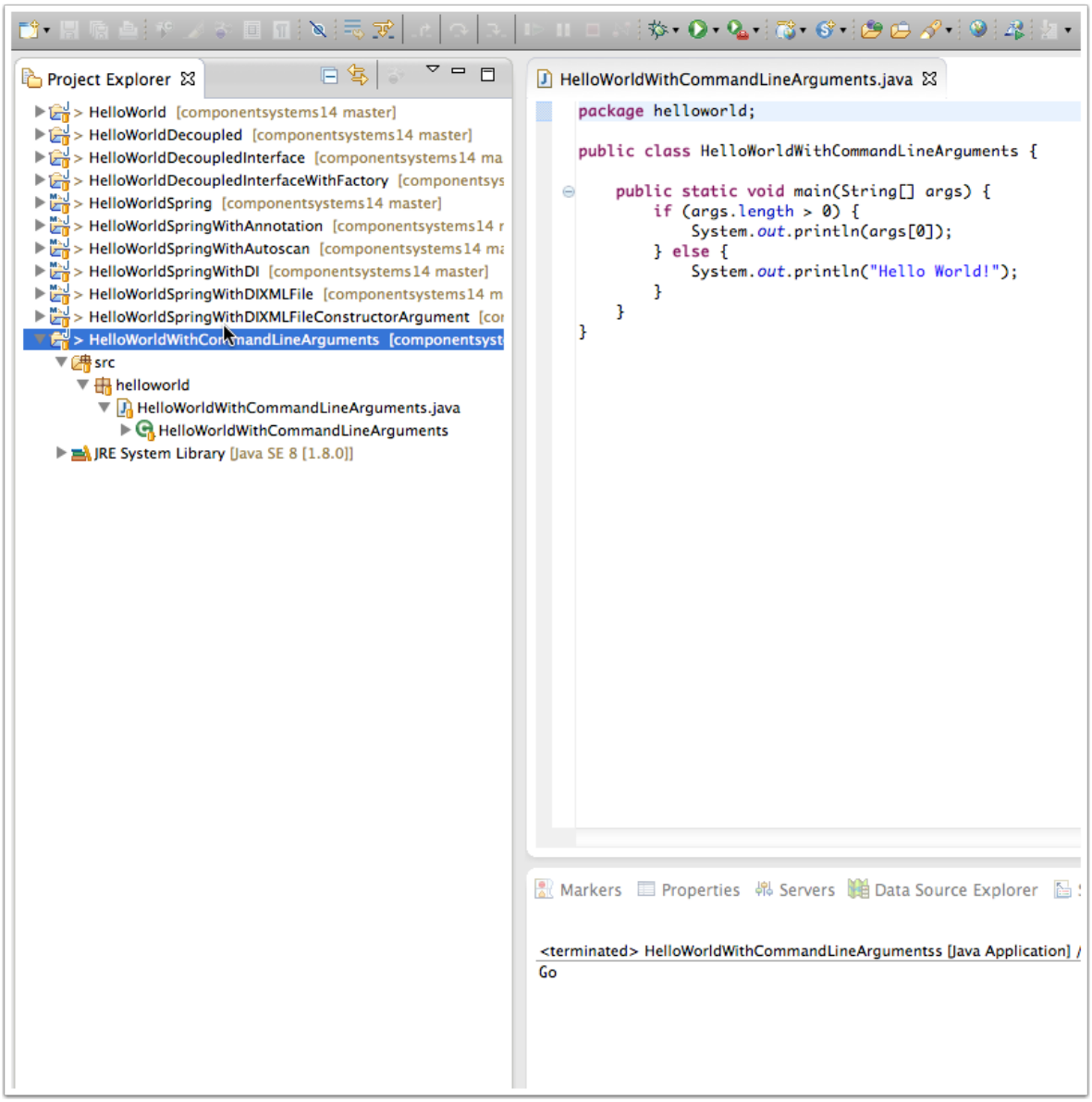
This code externalises the message content and read it in at runtime, from the command line argument. Now you can change the message without changing and recompiling the code.

1. Study `HelloWorldWithCommandLineArguments.java`

```
package helloworld;

public class HelloWorldWithCommandLineArguments {

  public static void main(String[] args) {
    if (args.length > 0) {
      System.out.println(args[0]);
    } else {
      System.out.println("Hello World!");
    }
  }
}
```

2. Run the application.

3. Observe that the text is displayed in the Console.



4 Build and Run `HelloWorldDecoupled` sample application

For this code, the message provider logic and the message renderer logic are separated from the rest of the code. So message provider (`HelloWorldMessageProvider`) can change without affecting the rest of the application (`StandardOutMessageRenderer`, `Launcher`) and the message renderer (`StandardOutMessageRenderer`) can change without affecting the rest of the application (`HelloWorldMessageProvider`, `Launcher`) .

1. Study the `HelloWorldDecoupled` sample application

HelloWorldDecoupled.java

```
package helloworld;
```

```
public class HelloWorldDecoupled {

  public static void main(String[] args) {
    StandardOutMessageRenderer mr = new StandardOutMessageRenderer();
    HelloWorldMessageProvider mp = new HelloWorldMessageProvider();
    mr.setMessageProvider(mp);
    mr.render();
  }
}
```

StandardOutMessageRenderer.java

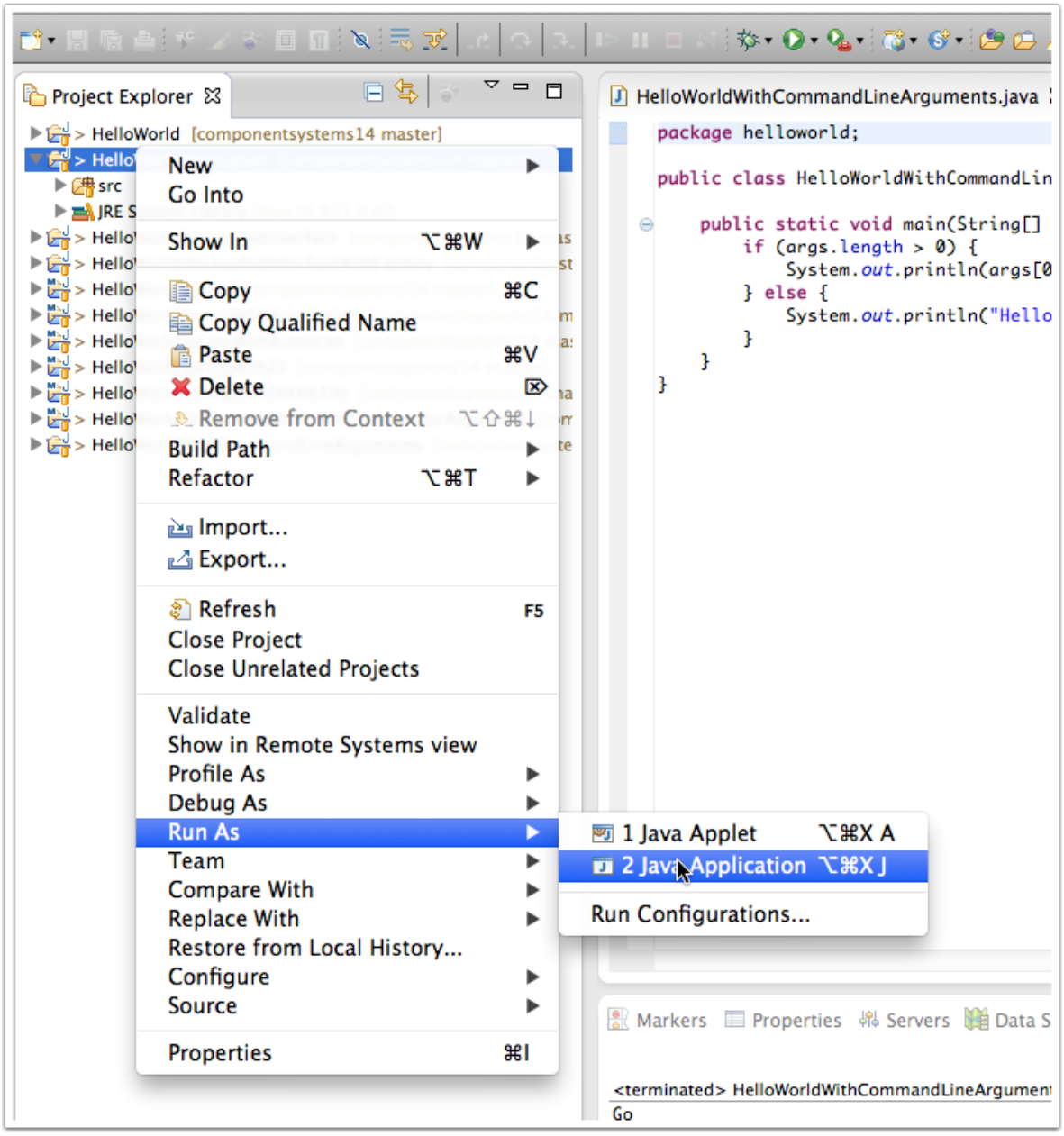
```
package helloworld;

public class StandardOutMessageRenderer {

  private HelloWorldMessageProvider messageProvider = null;

  public void render() {
    if (messageProvider == null) {
      throw new RuntimeException(
          "You must set the property messageProvider of class:"
              + StandardOutMessageRenderer.class.getName());
    }

    System.out.println(messageProvider.getMessage());
  }

  public void setMessageProvider(HelloWorldMessageProvider provider) {
    this.messageProvider = provider;
  }

  public HelloWorldMessageProvider getMessageProvider() {
    return this.messageProvider;
  }

}
```

HelloWorldMessageProvider.java

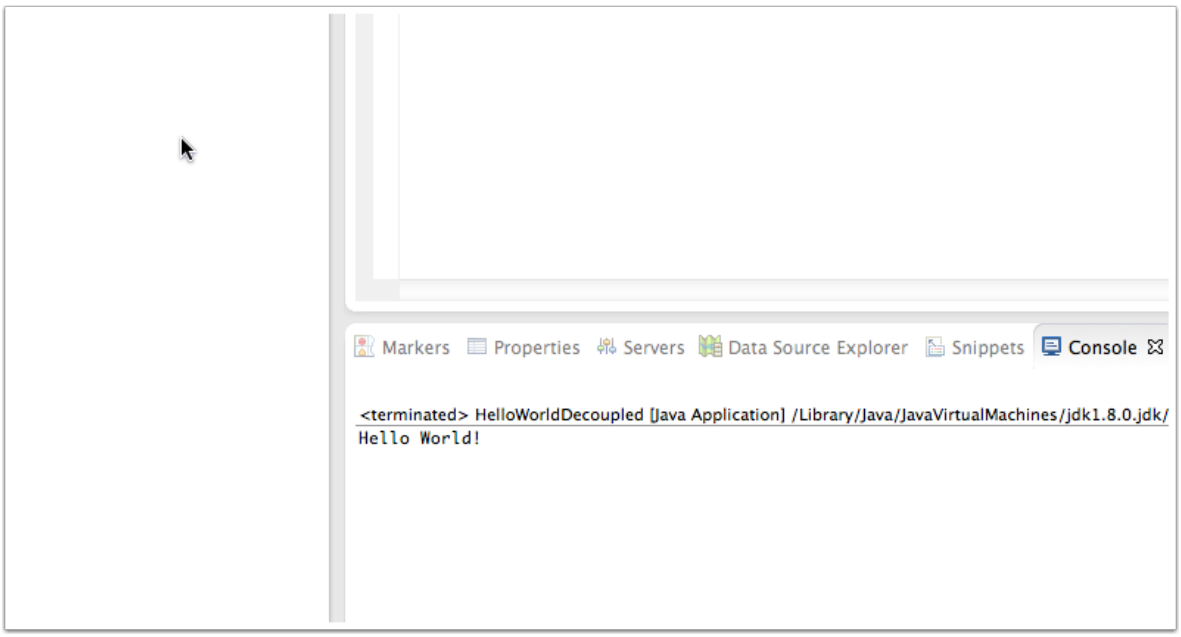
```
package helloworld;

public class HelloWorldMessageProvider {

  public String getMessage() {

    return "Hello World!";
```

```
  }

}
```

2. Run `HelloWorldDecoupled` sample application



3. Observe that the text is displayed in the Console.



5 Build and Run `HelloWorldDecoupledInterface` sample application

Now we are going to introduce Java interfaces so that the message rendering logic does not get affected by the change in message provider implementation.

1. Study the `HelloWorldDecoupledInterface` sample application

HelloWorldDecoupledInterface.java

```
package helloworld;

public class HelloWorldDecoupledInterface {

  public static void main(String[] args) {
    MessageRenderer mr = new StandardOutMessageRenderer();
    MessageProvider mp = new HelloWorldMessageProvider();
    mr.setMessageProvider(mp);
    mr.render();
  }
}
```

StandardOutMessageRenderer.java

```
package helloworld;

public class StandardOutMessageRenderer implements MessageRenderer {
```

```
  private MessageProvider messageProvider = null;

  public void render() {
    if (messageProvider == null) {
      throw new RuntimeException(
          "You must set the property messageProvider of class:"
              + StandardOutMessageRenderer.class.getName());
    }

    System.out.println(messageProvider.getMessage());
  }

  public void setMessageProvider(MessageProvider provider) {
    this.messageProvider = provider;
  }

  public MessageProvider getMessageProvider() {
    return this.messageProvider;
  }
}
```

MessageRenderer.java

```
package helloworld;

public interface MessageRenderer {

    public void render();

    public void setMessageProvider(MessageProvider provider);
    public MessageProvider getMessageProvider();
}
```

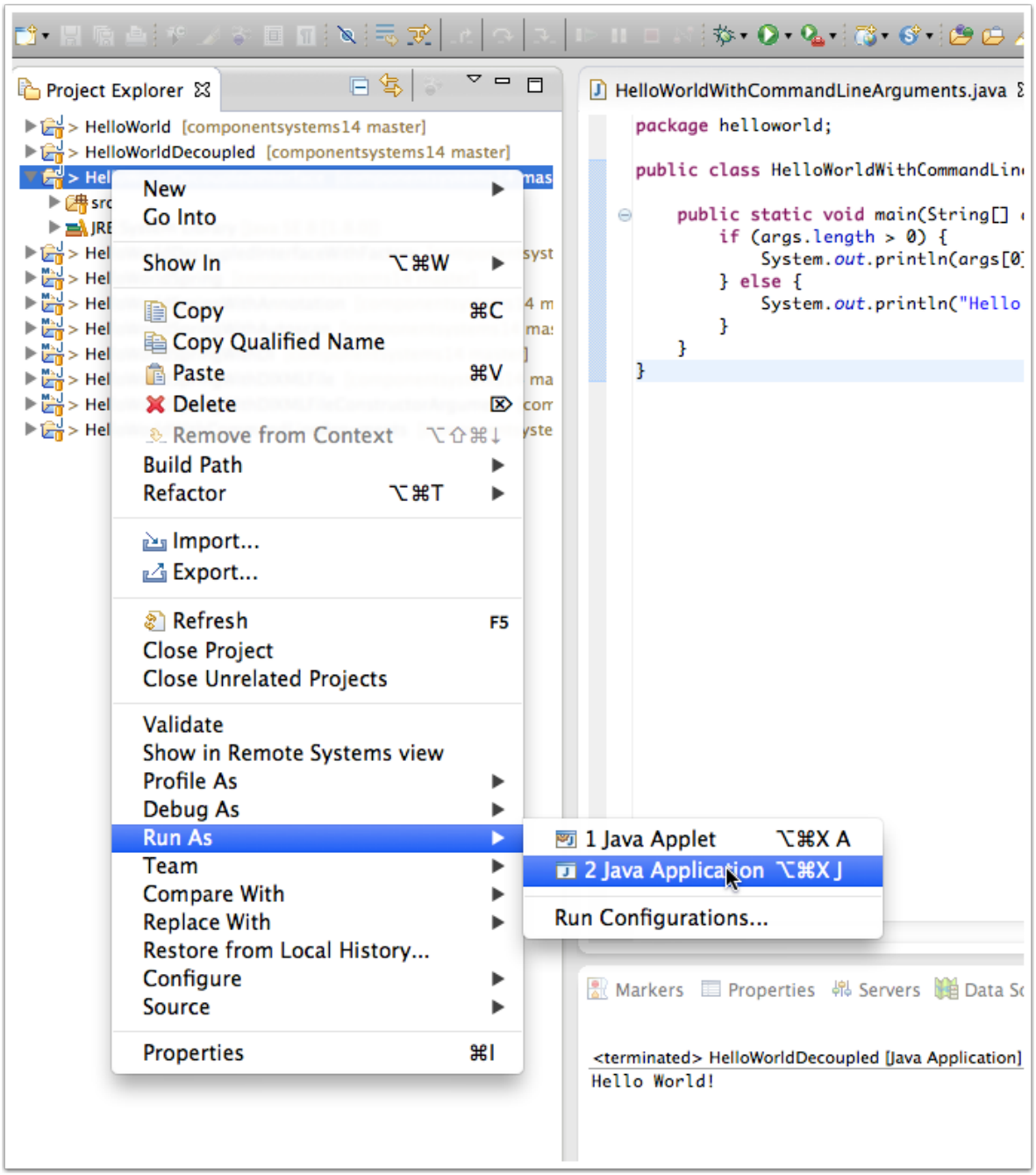
HelloWorldMessageProvider.java

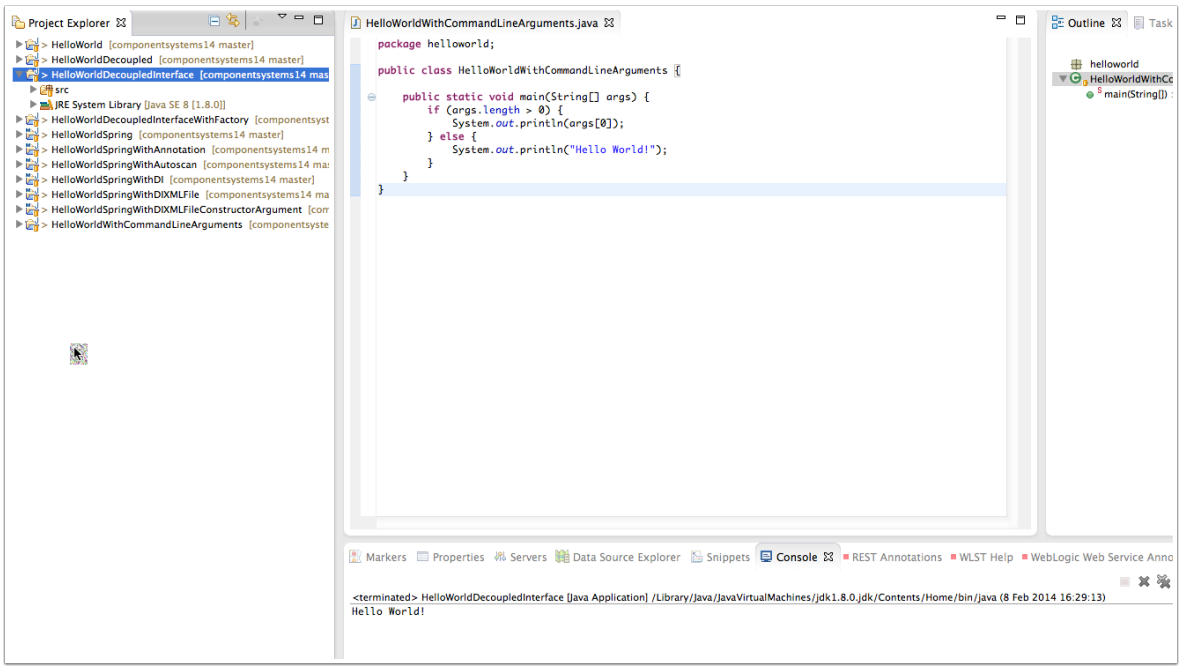
```
package helloworld;

public class HelloWorldMessageProvider implements MessageProvider {

  public String getMessage() {
    return "Hello World!";
  }

}
```

MessageProvider.java

```
package helloworld;

public interface MessageProvider {
  public String getMessage();
}
```

2. Run `HelloWorldDecoupledInterface` sample application





6 Build and Run `HelloWorldDecoupledInterfaceWithFactory` sample application

In this step, the properties file is used so that the message provider implementation and the message renderer implementation can be replaced simply by changing the properties file. So no change is required in the business logic code (launcher).

1. Study the `HelloWorldDecoupledInterface` sample application

HelloWorldDecoupledWithFactory.java

```
package helloworld;

public class HelloWorldDecoupledWithFactory {

  public static void main(String[] args) {
    MessageRenderer mr = MessageSupportFactory.getInstance()
        .getMessageRenderer();
    MessageProvider mp = MessageSupportFactory.getInstance()
        .getMessageProvider();
    mr.setMessageProvider(mp);
    mr.render();
  }
}
```

MessageSupportFactory.java

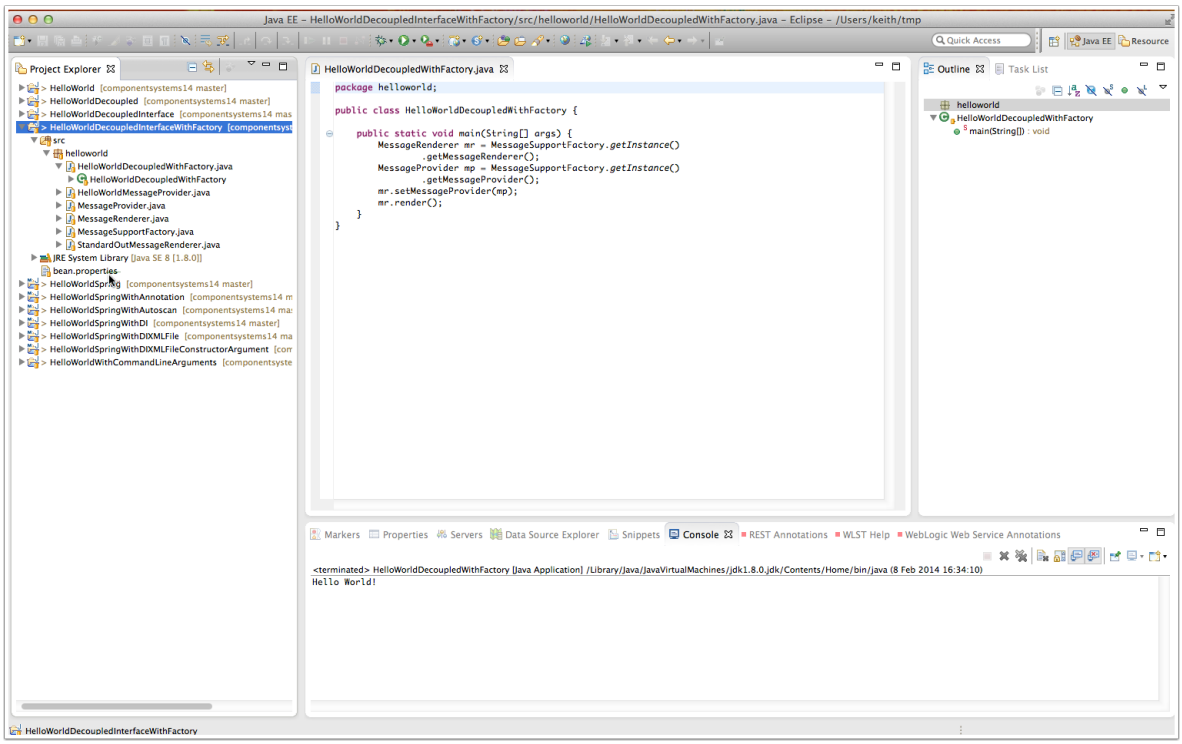
```
package helloworld;

import java.io.FileInputStream;
import java.util.Properties;

public class MessageSupportFactory {

  private static MessageSupportFactory instance = null;
  private Properties props = null;
  private MessageRenderer renderer = null;
  private MessageProvider provider = null;

  private MessageSupportFactory() {
    props = new Properties();

    try {
      props.load(new FileInputStream("bean.properties"));

      // get the implementation classes
      String rendererClass = props.getProperty("renderer.class");
      String providerClass = props.getProperty("provider.class");

      renderer = (MessageRenderer) Class.forName(rendererClass)
          .newInstance();
      provider = (MessageProvider) Class.forName(providerClass)
          .newInstance();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  static {
    instance = new MessageSupportFactory();
  }

  public static MessageSupportFactory getInstance() {
    return instance;
  }

  public MessageRenderer getMessageRenderer() {
    return renderer;
  }

  public MessageProvider getMessageProvider() {
    return provider;
```

```
  }

}
```

bean.properties

```
renderer.class=helloworld.StandardOutMessageRenderer
provider.class=helloworld.HelloWorldMessageProvider
```

2. Run `HelloWorldDecoupledInterfaceWithFactory` sample application



7 Build and Run `HelloWorldSpring` sample application

Now you are going to remove the need of your own *glue code* (`MessageSupportFactory`) and use the Spring provided factory class instead.

1. Study the `HelloWorldSpring` sample application

HelloWorldSpring.java

```
package helloworld;

import java.io.FileInputStream;
import java.util.Properties;
```

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;

public class HelloWorldSpring {

    public static void main(String[] args) throws Exception {

        // get the bean factory
        BeanFactory factory = getBeanFactory();

        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        MessageProvider mp = (MessageProvider) factory.getBean("provider");

        mr.setMessageProvider(mp);
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        // get the bean factory - understanding DefaultListableBeanFactory(0
      // not really important.  It is just an Factory class example from
      // Spring.
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

        // create a definition reader
        PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(
                factory);

        // load the configuration options
        Properties props = new Properties();
        props.load(new FileInputStream("beans.properties"));

        rdr.registerBeanDefinitions(props);

        return factory;
    }
}
```

beans.properties

```
renderer.(class)=helloworld.StandardOutMessageRenderer
provider.(class)=helloworld.HelloWorldMessageProvider
```

2. Run `HelloWorldSpring` sample application

8 Build and Run `HelloWorldSpringWithDI` sample application

In this step you are going to use the Spring framework Dependency Injection (DI) mechanism. The `main()` method now just obtains the `MessageRenderer` bean and calls `render()`. It does not have to obtain the `MessageProvider` bean and set the `MessageProvider` property of the `MessageRenderer` bean itself because the *wiring* is performed through the Spring framework's Dependency Injection mechanism.

1. Study the `HelloWorldSpringWithDI` sample application

HelloWorldSpringWithDI.java

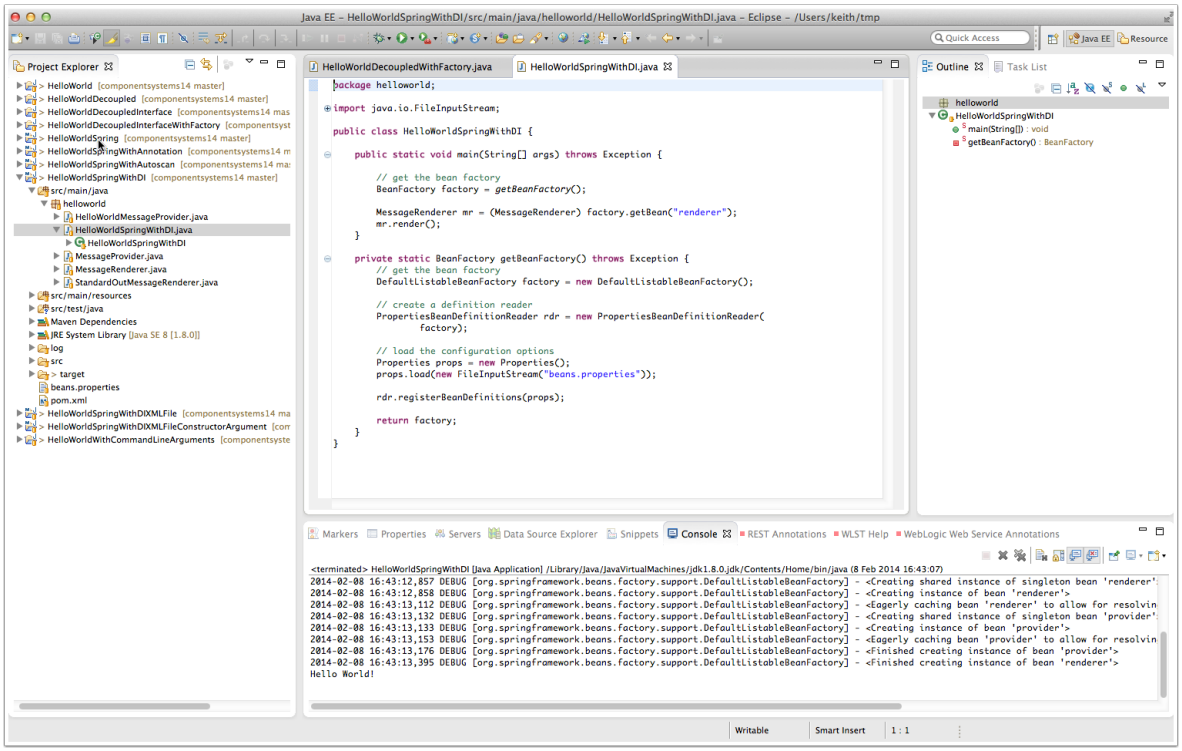
```
package helloworld;

import java.io.FileInputStream;
import java.util.Properties;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;

public class HelloWorldSpringWithDI {

    public static void main(String[] args) throws Exception {

        // get the bean factory
        BeanFactory factory = getBeanFactory();

        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        // get the bean factory
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

        // create a definition reader
        PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(
                factory);

        // load the configuration options
        Properties props = new Properties();
        props.load(new FileInputStream("beans.properties"));

        rdr.registerBeanDefinitions(props);

        return factory;
```

```
    }
}
```

beans.properties

```
#Message renderer
renderer.(class)=helloworld.StandardOutMessageRenderer
#  Ask Spring to assign provider bean to the MessageProvider property
#  of the Message renderer bean
renderer.messageProvider(ref)=provider

#Message provider
provider.(class)=helloworld.HelloWorldMessageProvider
```

2. Run `HelloWorldSpringWithDI` sample application



9 Build and Run `HelloWorldSpringWithDIXMLFile` sample application

In this step, you are going to use an XML file in which the wiring requirements are specified.

1. Study the `HelloWorldSpringWithDIXMLFile` sample application

HelloWorldSpringWithDIXMLFile.java

```
package helloworld;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class HelloWorldSpringWithDIXMLFile {

  public static void main(String[] args) throws Exception {

    // get the bean factory
    BeanFactory factory = getBeanFactory();
    MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
    mr.render();
  }

  private static BeanFactory getBeanFactory() throws Exception {
    // create a bean factory from beans.xml
    BeanFactory factory = new ClassPathXmlApplicationContext("/beans.xml");
    return factory;
  }
}
```

2. Run `HelloWorldSpringWithDIXMLFile` sample application

10 Build and Run `HelloWorldSpringWithDIXMLFileConstructorArgument` sample application

In this step, you are going to change the wiring requirement using the constructor.

1. Study the `HelloWorldSpringWithDIXMLFileConstructorArgument` sample application

HelloWorldSpringWithDIXMLFileConstructorArgument.java

```
package helloworld;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class HelloWorldSpringWithDIXMLFileConstructorArgument {

    public static void main(String[] args) throws Exception {

        // get the bean factory
        BeanFactory factory = getBeanFactory();
```

```
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        // create a bean factory from beans.xml
        BeanFactory factory = new ClassPathXmlApplicationContext("/beans.xml");
        return factory;
    }
}
```

beans.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-3.0.xsd">

    <bean id="renderer" class="helloworld.StandardOutMessageRenderer">
        <property name="messageProvider"
                  ref="provider"/>
    </bean>
    <bean id="provider" class="helloworld.ConfigurableMessageProvider">
        <constructor-arg>
            <value>This is a configurable message</value>
        </constructor-arg>
    </bean>
</beans>
```

ConfigurableMessageProvider.java

```
package helloworld;

public class ConfigurableMessageProvider implements MessageProvider {

    private String message;

    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
```

```
}
```

2. Run `HelloWorldSpringWithDIXMLFileConstructorArgument` sample application

11 Build and Run `HelloWorldSpringWithAnnotation` sample application

In this step, you are going to use Java annotations as the injection mechanism.

1. Study the code

StandardOutMessageRenderer.java

```
package helloworld;


public class StandardOutMessageRenderer implements MessageRenderer {

    private MessageProvider messageProvider = null;

    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException(
                    "You must set the property messageProvider of class:"
                            + StandardOutMessageRenderer.class.getName());
        }

        System.out.println(messageProvider.getMessage());
    }

    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;
    }

    public MessageProvider getMessageProvider() {
        return this.messageProvider;
    }

}
```

beans.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-3.0.xsd">

    <bean id="renderer" class="helloworld.StandardOutMessageRenderer">
        <property name="messageProvider"
                  ref="provider"/>
    </bean>
    <bean id="provider" class="helloworld.ConfigurableMessageProvider">
        <constructor-arg>
            <value>This is a configurable message</value>
        </constructor-arg>
    </bean>
</beans>
```

2. Run `HelloWorldSpringWithAnnotation` sample application

12 Build and Run `HelloWorldSpringWithAutoscan` sample application

In this step, you are going to use the *autoscan* feature of Spring DI.

1. Study the code

beans.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"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- enable the usage of annotations -->
    <!--  <context:annotation-config /> -->

    <!-- scan component, it also assumes annotation-config as well-->
    <context:component-scan base-package="helloworld"/>
</beans>
```

StandardOutMessageRenderer.java

```
package helloworld;

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

@Component("renderer")   // This is the same as @Component(value="renderer")
public class StandardOutMessageRenderer implements MessageRenderer {

  @Autowired
    private MessageProvider messageProvider = null;

    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException(
                    "You must set the property messageProvider of class:"
                            + StandardOutMessageRenderer.class.getName());
        }

        System.out.println(messageProvider.getMessage());
    }
}
```

HelloWorldMessageProvider.java

```
package helloworld;

import org.springframework.stereotype.Component;

@Component
public class HelloWorldMessageProvider implements MessageProvider {

    public String getMessage() {

        return "Hello World! --- with Autoscan! How does that work?";
    }

}
```

2. Run `HelloWorldSpringWithAutoscan` sample application