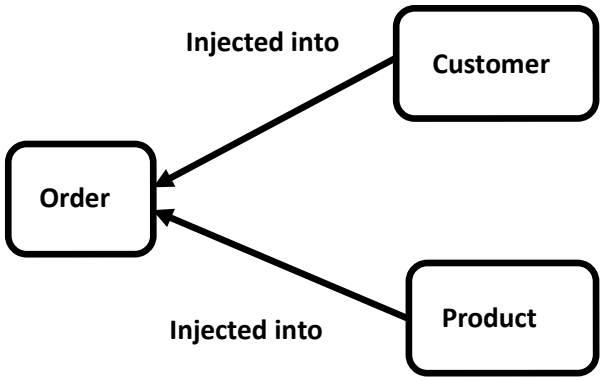
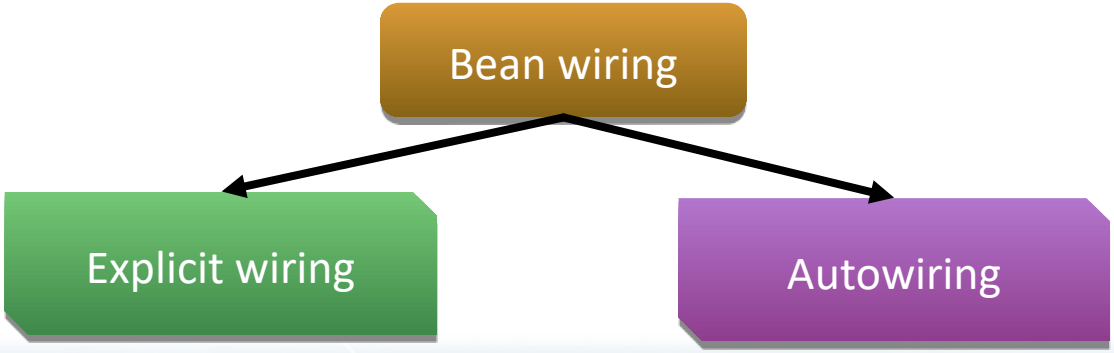
# Building Web Applications Using the Spring Framework

Aesthetic: Tarkeshwar Barua

# Injecting Application Objects

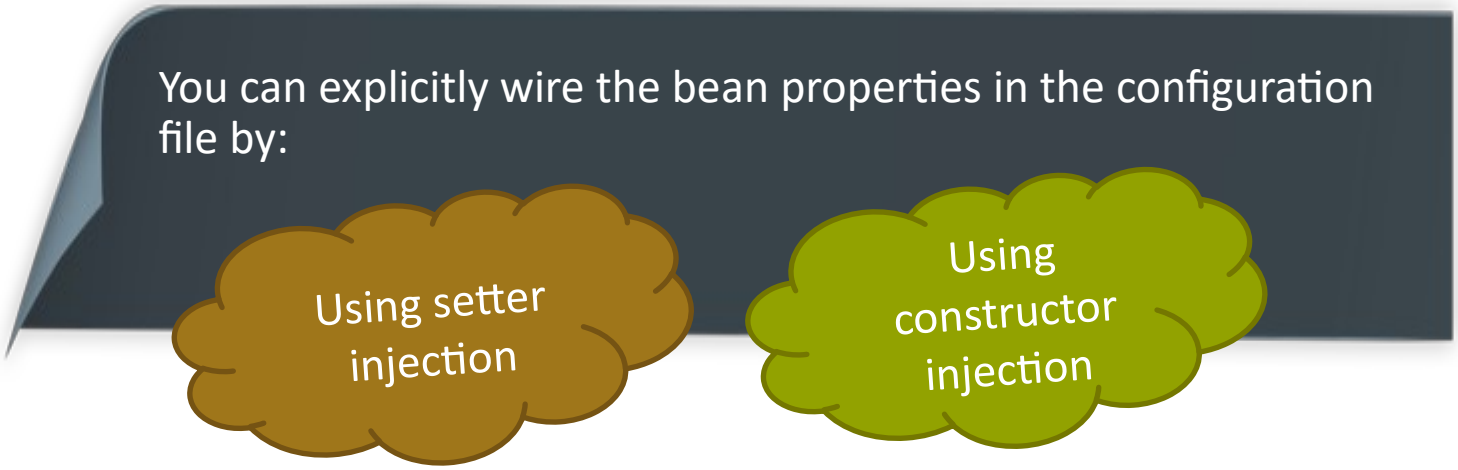



- ***The process of creating and managing association among application objects forms the core of DI and is commonly referred to as wiring.***
- ***To inject dependencies into the application objects, you need to configure them in an XML-based configuration file.***
- ***In this file, you should first give a definition for the bean for which the dependency is to be created.***



# Applying Explicit Wiring

- ***You can explicitly wire the beans by declaring their dependencies in the configuration file.***
- ***While declaring the beans in the configuration file, you can also wire its properties.***



# Applying Explicit Wiring (Contd.)


- ***Setter injection is a bean wiring technique in which the JavaBean setter methods are used for supplying the bean properties to the objects that need them.***
- ***You can wire the bean properties in a Spring-enabled application by declaring a `<property>` element for the bean in the configuration file within the `<bean>` tag.***

The `<property>` element can inject dependencies using the setter methods of the property in the following ways:


# Applying Explicit Wiring (Contd.)


- ***You can use the `value` attribute of the `<property>` element to inject a string value into a bean property.***
- ***The `value` attribute can also take on values of the other types, such as integer, floating point, and boolean.***

Example: `Player` interface

```
package SetterInject;
public interface Player
{
    void play();
}
```

# Applying Explicit Wiring (Contd.)

Example: **FootballPlayer** class that implements the **Player** interface

```
package SetterInject;
public class FootballPlayer implements Player{
 public FootballPlayer() {}
 @Override
 public void play() {
 System.out.println("I am playing with " + football +
 " football.");
  }
 private String football;
 public void setFootball(String football){
 this.football = football;
 }
 private FootballBoots boots;
 public void setBoots(FootballBoots boots){
 this.boots = boots;
  }
}
```

# Applying Explicit Wiring (Contd.)

Example: The **`FootballBoots`** interface

```
package SetterInject;
public interface FootballBoots
{
  void wearBoots();
}
```

Example: Bean wiring

```
<bean id="Kaka"
class="SetterInject.FootballPlayer">
<property name="football" value="Adidas"/>
</bean>

<bean id="Forlan"
class="SetterInject.FootballPlayer">
<property name="football" value="Nike"/>
</bean>
```

# Applying Explicit Wiring (Contd.)


- ***To wire beans with each other, you have to specify bean references in the bean configuration file.***
- ***You can specify a bean reference for a bean property by using the `ref` attribute of the `<property>` element.***
- ***The `ref` attribute sets the value of the specified property of the bean by passing to it a reference of another bean.***

Example: `FootballBoots` class

```
package SetterInject;
public class Predator implements
FootballBoots {
  public Predator () {}
  public void wearBoots () {
    System.out.println("I am wearing
Adidas predator boots.");
  }
}
```

# Applying Explicit Wiring (Contd.)

Example: Bean configuration

```
<bean id="predator"
class="SetterInject.Predator"/>
```

Example: Bean wiring

```
<bean id="Kaka"
class="SetterInject.FootballPlayer">
<property name="football" value="Adidas" />
<property name="boots" ref="predator" />
</bean>

<bean id="Forlan"
class="SetterInject.FootballPlayer">
<property name="football" value="Nike" />
<property name="boots" ref="predator" />
</bean>
```

# Applying Explicit Wiring (Contd.)


- ***It is a bean wiring technique, where an object is provided its dependencies via its own constructors.***
- ***The dependencies are passed as arguments of the constructors with each representing a property or a collaborator object.***
- ***In this method, each object declares a constructor or a set of constructors that take object dependencies as arguments.***

# Applying Explicit Wiring (Contd.)

Example: Updated `FootballPlayer` class

```
package SetterInject;
public class FootballPlayer implements Player{

 private int shirtNumber;

 public FootballPlayer() {}

 public FootballPlayer(int shirtNumber){
   this.shirtNumber= shirtNumber;
 }
 @Override
 public void play() {

    System.out.println("I am playing with shirt
number " + shirtNumber +".");
 }
}
```

# Applying Explicit Wiring (Contd.)

Example: Bean wiring

```
<bean id="Kaka"
 class="SetterInject.FootballPlayer">
 <constructor-arg value="7"/>
</bean>

<bean id="Forlan"
 class="SetterInject.FootballPlayer">
 <constructor-arg value="21"/>
</bean>
```

# Applying Autowiring


- ***The property name or the constructor argument is not declared within the configuration file.***
- ***The Spring container itself finds the type and name of the property and matches the property type and name with other beans in the container based on their specified type or name.***
- ***In this method, each object declares a constructor or a set of constructors that take object dependencies as arguments.***
- ***Wiring is achieved by setting the `autowire` property of the beans.***

# Applying Autowiring (Contd.)

**Values of the `autowire` property**

**`byName`**

Bean is matched whose name matches the name of the property being wired.

**`byType`**

Bean is matched whose type matches the type of the property being wired.

**`constructor`**

Bean is matched based on the parameters of the constructors of the bean that is being wired.

**`autodetect`**

Bean is matched first by using `constructor`, and then `byType`, if there is a default constructor with no arguments.

# Applying Autowiring (Contd.)


- ***If the name of a property matches the name of the bean that has to be wired into that property, the Spring framework can automatically wire that bean into the property.***

Example: Bean wiring

```
<bean id="Kaka"
class="SetterInject.FootballPlayer">
<property name="football" value="Adidas"/>
<property name="boots" ref="predator"/>
</bean>
```

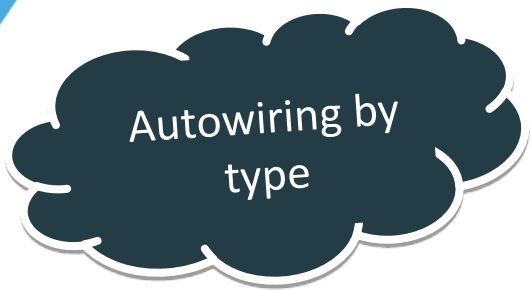
Example: Bean configuration

```
<bean id="boots"
class="SetterInject.Pred
ator"/>
```

Example: Bean autowiring

```
<bean id="Kaka"
class="SetterInject.FootballPlayer"
autowire="byName">
<property name="football" value="Adidas" />
</bean>
```

# Applying Autowiring (Contd.)




- ***Spring attempts to find a bean whose type is compatible with the property type.***

Example:

```
<bean id="boots"
class="SetterInject.Predator"/>
<bean id="Kaka"
class="SetterInject.FootballPlayer"
autowire="byType">
<property name="football" value="Adidas" />
</bean>
```

# Applying Autowiring (Contd.)

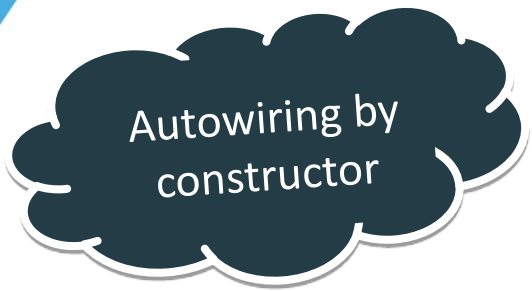
Autowiring by constructor

- ***Is achieved by setting the `autowire` property to `constructor`.***
- ***Spring automatically selects constructor arguments from the beans, defined in the configuration file.***

Example:

RugbyPlayer&RugbyBootsClasses

# Applying Autowiring (Contd.)

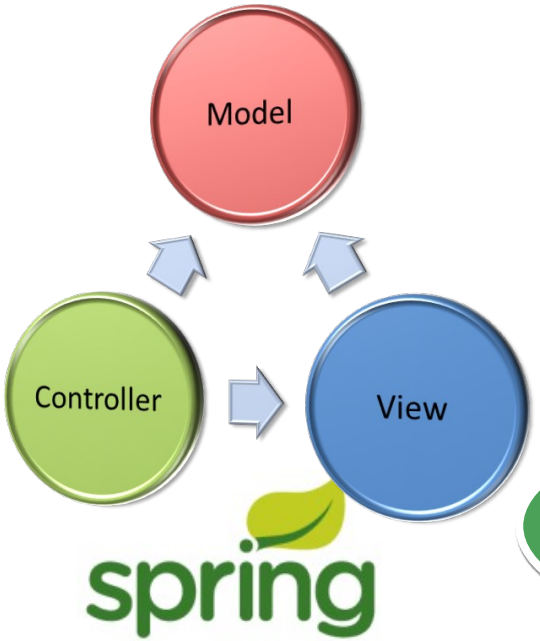



Example: Bean wiring

```
<bean id="Wilkinson"
class="AutowireInject.RugbyPlayer"
autowire="constructor">
<property name="shirtNumber" value="7"/>
</bean>

<bean id="NikeBoots"
class="AutowireInject.RugbyBoots">
<property name="boots" value="Nike"/>
</bean>
```

# Exploring Spring MVC

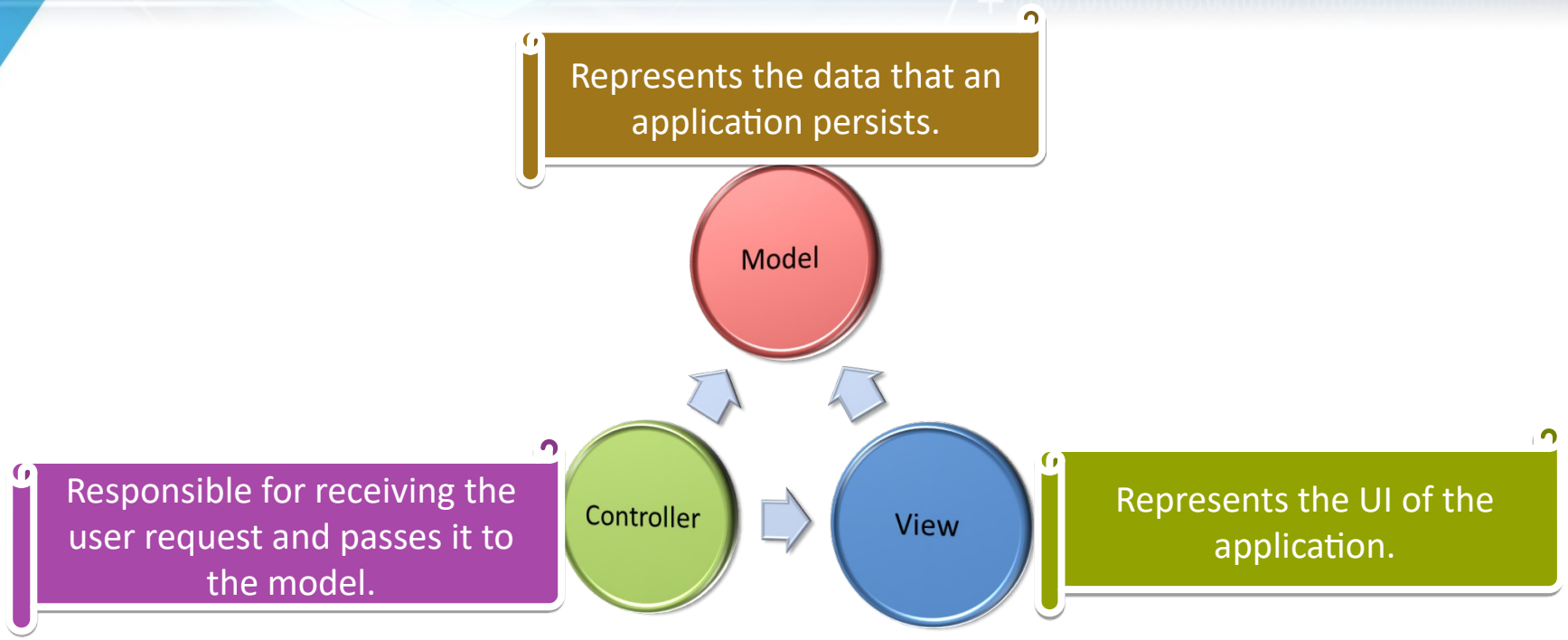



- ***Is an MVC design pattern extension that allows you to represent the UI flow of a Web application into individual controllers and views.***

## Features

- Pluggable view technology
- Injection of services into controllers
- Integration support

# Introducing Spring MVC Framework

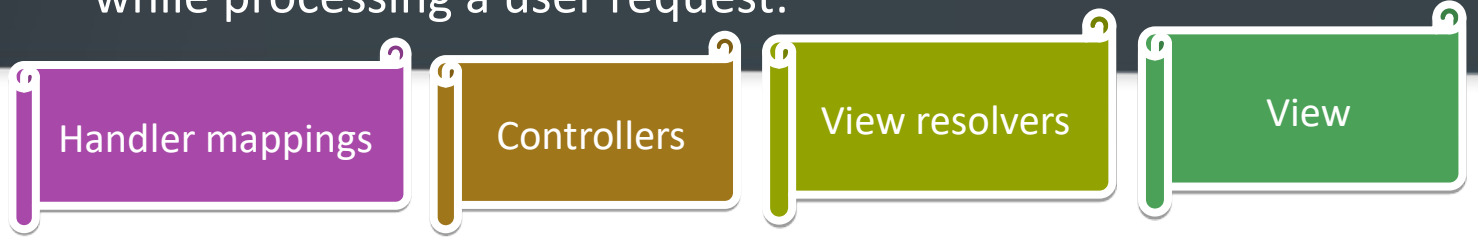



- ***The core idea of the MVC pattern is to separate the business logic from UIs to allow them to change independently without affecting each other.***
- ***The Spring Web MVC framework advantage of the AOP and DI features of the Spring framework to help you create loosely coupled Web applications.***

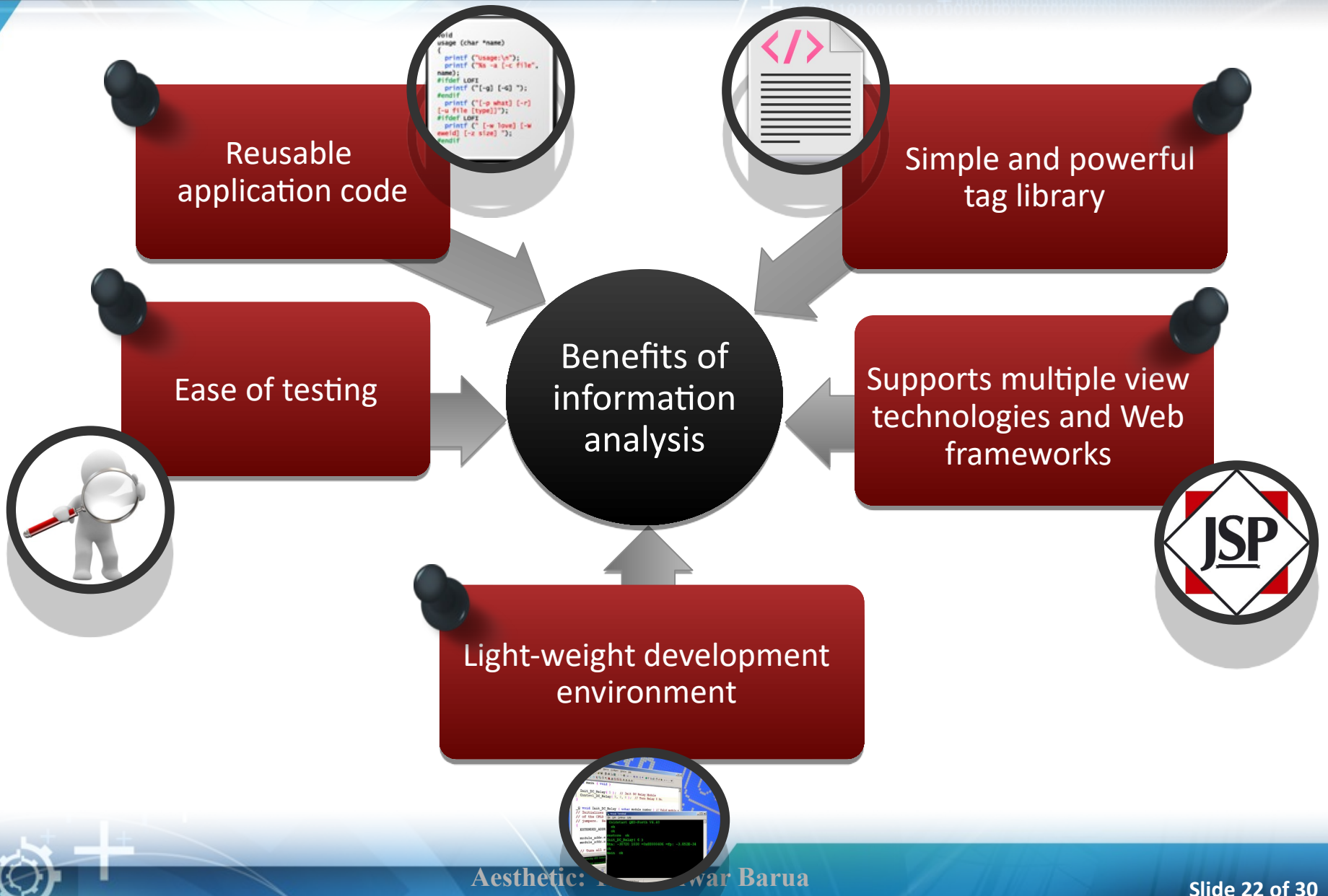
# Introducing Spring MVC Framework (Contd.)

- ***The Spring Web MVC framework is built around a front controller servlet called dispatcher servlet.***
- ***The dispatcher servlet is responsible for delegating the user request to various components of the application while executing a user request.***

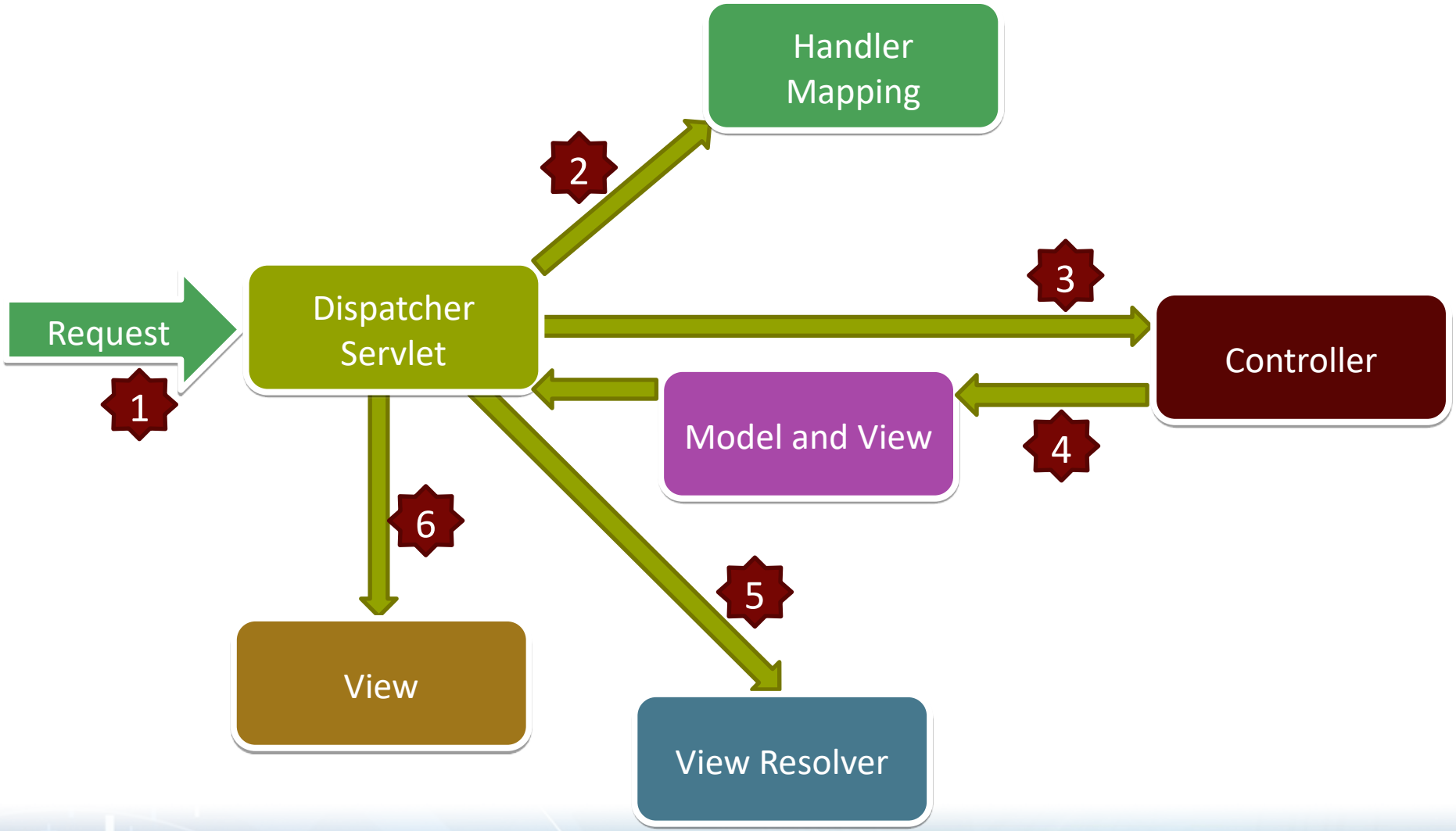
The Spring MVC framework makes use of the following components while processing a user request:



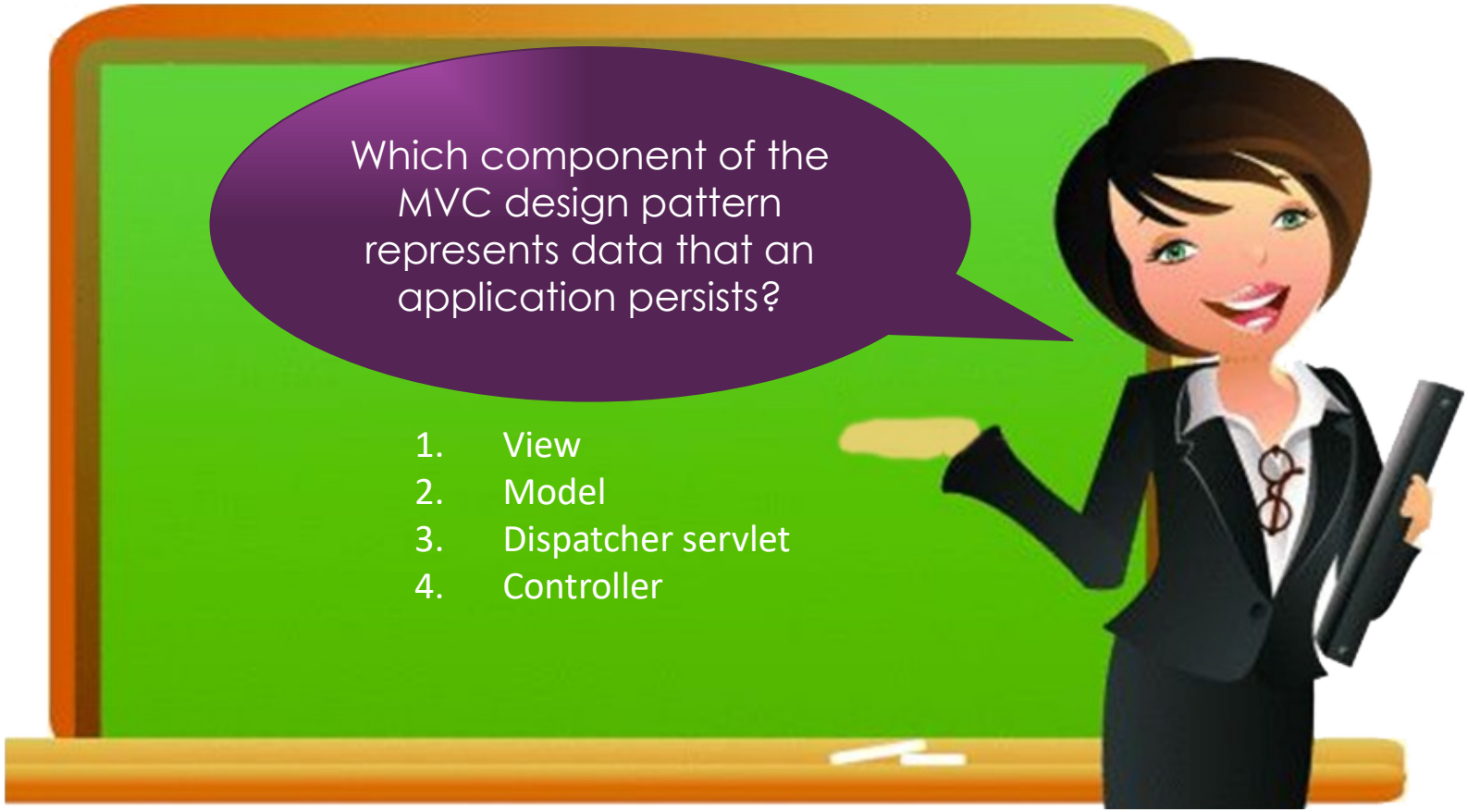
# Benefits of the Spring MVC Framework



# Life Cycle of a Web Request



# Just a Minute




Answer: 2. Model

# Handling Web Requests

Steps to handle Web requests:

1. Create and configure a dispatcher servlet.
2. Create the controller class that performs the business logic for the requested page.
3. Configure the controller within the dispatcher servlet's context configuration file.
4. Declare a view resolver to tie the controller with the view.
5. Create a view to render the requested page to the user.

# Creating and Configuring a Dispatcher Servlet

- ***The dispatcher servlet intercepts all user requests before passing them to a controller class.***

- ***Declare and configure the dispatcher servlet in the `web.xml` file with the help of the `<servlet>` and `<servlet-mapping>` elements.***

Example:

```
<servlet>
<servlet-name>dispatcher</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
```

# Creating and Configuring a Dispatcher Servlet

- ***You may download all required jar files from the given source***

***https://static.javatpoint.com/src/sp/springjars.zip***

Example:

```
<servlet>
<servlet-name>dispatcher</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
```

# Creating and Configuring a Dispatcher Servlet (Contd.)

- ***You can use the <servlet-mapping> element to specify which URLs will be handled by the dispatcher servlet.***

Example:

```
<servlet-mapping>
<servlet-name>dispatcher</servlet-name>
<url-pattern>*.htm</url-pattern>
</servlet-mapping>
```