

UNIT 3

JavaBean

Introduction to JavaBean:

“A JavaBean is a reusable software component that can be manipulated visually in a builder tool.”

They are classes that encapsulate many objects into a single object (the bean). Following are the unique characteristics that distinguish a JavaBean from other Java classes –

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the **Serializable** interface.
- It may have a number of properties which can be read or written.
- It may have a number of "getter" and "setter" methods for the properties.

Why use JavaBean:

- **Java Bean** is a reusable software component and is easy to maintain.
- Presence of a no argument constructor;
- Support for persistence;
- Properties manipulated by getter and setter methods;
- Support for introspection;(Introspection is the automatic process of analysing a bean's design patterns to reveal the bean's properties, events, and methods. This process controls the publishing and discovery of bean operations and properties.)
- Events as the mechanism of communication between beans;
- Support for customization via property editors.

Advantages of JavaBean:

- Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to an application builder tool can be controlled.
- A Bean may be designed to operate correctly in different locales, which makes it useful in global markets.
- Auxiliary software can be provided to help a person configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.

- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Disadvantages of JavaBean:

- A class with a nullary constructor is subject to **being instantiated in an invalid state**. If such a class is instantiated manually by a developer (rather than automatically by some kind of framework), the developer might not realize that the class has been improperly instantiated.
- The compiler can't detect such a problem, and even if it's documented, there's no guarantee that the developer will see the documentation.
- Having to create a getter for every property and a setter for many, most, or all of them creates an **immense amount of boilerplate code**.
- **Boilerplate code** are sections of code that have to be included in many places with **little or no alteration**. When using languages that are considered verbose, **the programmer must write a lot of code to accomplish only minor functionality**. Such code is called boilerplate.

JavaBean Properties:

A JavaBean property is a named attribute that can be **accessed by the user of the object**.

The attribute can be of any Java data type, including the classes that you define.

A JavaBean property may be **read, write, read only, or write only**.

JavaBean properties are accessed through two methods in the JavaBean's implementation class –

Sr. No.	Method and Description
1	getPropertyName() For example, if property name is <i>firstName</i> , your method name would be getFirstName() to read that property. This method is called accessor .
2	setPropertyName() For example, if property name is <i>firstName</i> , your method name would be setFirstName() to write that property. This method is called mutator .

getter method:

- It should be public in nature.
- The return type should not be void i.e. according to our requirement we have to give return-type.
- The getter method should be prefixed with get
- It should not take any argument.

no arg

setter method:

- It should be public in nature.
- The return type should not be void
- The setter method should be prefixed with set
- It should take some argument i.e. it should not be no-arg method.

arg

JavaBean Example:

```
package JavaBean;
```

```
public class StudentsBean implements java.io.Serializable {
```

```
    private String firstName = null;
```

```
    private String lastName = null;
```

```
    private int age = 0;
```

```
    public StudentsBean() {  
    }
```

```
    public String getFirstName(){  
        return firstName;  
    }
```

```
    public String getLastName(){  
        return lastName;  
    }
```

```
    public int getAge(){  
        return age;  
    }
```

```
    public void setFirstName(String firstName){  
        this.firstName = firstName;  
    }
```

```
    public void setLastName(String lastName){  
        this.lastName = lastName;  
    }
```

```
    public void setAge(Integer age){  
        this.age = age;  
    }
```

```
}
```

should have arguments

Accessing JavaBeans:

The `useBean` action declares a `JavaBean` for use in a `JSP`. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the `JSP`. The full syntax for the `useBean` tag is as follows

```
<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>

<html>
  <head>
    <title>get and set properties Example</title>
  </head>

  <body>
    <jsp:useBean id = "students" class = " JavaBean.StudentsBean">
      <jsp:setProperty name = "students" property = "firstName" value =
"Zara"/>
      <jsp:setProperty name = "students" property = "lastName" value = "Ali"/>
      <jsp:setProperty name = "students" property = "age" value = "10"/>
    </jsp:useBean>

    <p>Student First Name:
      <jsp:getProperty name = "students" property = "firstName"/>
    </p>

    <p>Student Last Name:
      <jsp:getProperty name = "students" property = "lastName"/>
    </p>

    <p>Student Age:
      <jsp:getProperty name = "students" property = "age"/>
    </p>

  </body>
</html>
```

Output:

Student First Name: Zara

Student Last Name: Ali

Student Age: 10

How to access JavaBean Class using Java Application

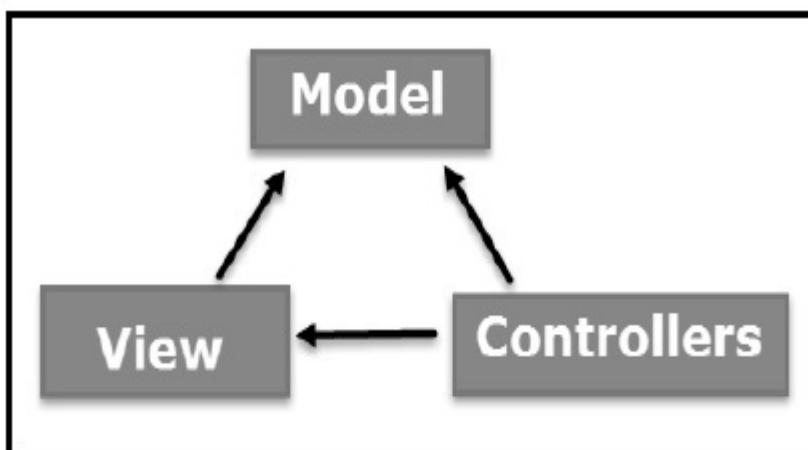
```
package JavaBean;  
public class Test  
{  
    public static void main(String args[])  
    {  
        StudentsBean s = new StudentsBean (); // object is created  
        s.setfirstName ("Zara");  
        s.setlastName ("Ali");  
        s.setage ("10"); // setting value to the object  
        System.out.println(s.getfirstName ());  
        System.out.println(s.getlastName ());  
        System.out.println(s.getage ());  
    }  
}
```

Struts 2

Introduction to MVC Framework:

The **Model-View-Controller (MVC)** is an architectural pattern that separates an application into three main logical components: the **model**, the **view**, and the **controller**.

Each of these components are built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.



- **Model**

1. The Model component corresponds to all the **data-related logic** that the user works with.
2. This can represent either the **data that is being transferred between the View and Controller** components or any other business logic-related data.
3. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

- **View**

1. The View component is used for all the **UI logic** of the application.
2. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

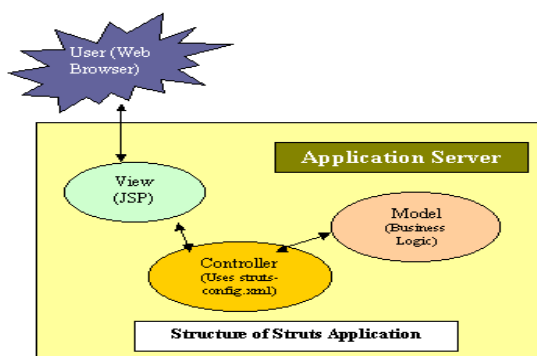
- **Controller**

1. Controllers act as an **interface between Model and View** components to process all the business logic and incoming requests, **manipulate data** using the Model component and **interact with the Views** to render the final output.
2. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.

What is **Strut2**?

Struts framework helps for developing the web-based applications. Struts java framework is one of the most popular framework for web based applications. Java servlet, JavaBeans, ResourceBundles and XML etc are the Jakarta commons packages used for accomplishing this purpose.

This is an **open source implementation of MVC pattern** for the development of web-based application.



The features of this type of framework are,

- More **robust** or **reliable** architecture
- Helps for development of application of any size
- Easy to design
- **Scalable**
- Reliable web application with java.

Struts 2 Framework Features

1. A framework is a group of services that provide developers with **common set of functionality** to be reused amongst multiple applications
2. Struts is a java based framework which separate the application logic that interacts with the database from an HTML page that form the response
3. Struts is not a technology, **it's a framework** that can be used along with other java based technologies
4. Struts make ease of enterprise application development with flexible and extensible application architecture and custom tags
5. **Struts 2 is a web application framework** based on the OpenSymphony WebWork framework that **implements MVC design pattern**

1) Configurable MVC components

- a) In struts 2 framework, we provide all the components (view components and action) information in **struts.xml file**. ← .xml
- b) If we need to change any information, we can simply change it in the xml file.

2) POJO based actions

- a) In struts 2, action class is **POJO (Plain Old Java Object)** i.e. a simple java class.
- b) Here, you are not forced to implement any interface or inherit any class.

3) AJAX support

- a) Struts 2 provides support to ajax technology. It is used to make asynchronous request i.e. it doesn't block the user.
- b) It sends only required field data to the server side not all. So it **makes the performance fast**.

4) Integration Support

- a) We can simply integrate the struts 2 application with **hibernate, spring, tiles** etc. frameworks.

5) Various Result Types

- a) We can use **JSP, freemarker, velocity** etc. technologies as the result in struts 2.

6) Various Tag support

- a) Struts 2 provides various types of tags such as UI tags, Data tags, control tags etc to ease the development of struts 2 application.

7) Theme and Template support

- a) Struts 2 provides three types of theme support: xhtml, simple and css_xhtml.
- b) The xhtml is default theme of struts 2. Themes and templates can be used for common look and feel.

Components of Struts 2 framework:

1. Actions

Model

- a) Model is implemented in struts using actions.
- b) Actions include the business logic and interacts with persistence storage ,retrieve and manipulate data.

2. Interceptors

- a) Interceptor is an object, that is invoked (executed) the pre-processing request and post-processing request.
- b) Interceptors are used to perform the validations, exceptions handling and display the result in between the execute() method.
- c) In struts 2, Interceptors execution is same as the servlet filter execution type.
- d) Using Interceptors, One can run the application using validations, and can remove the validations on that application and one cannot deploy the application in the server. If it directly run on the server.

3. Value Stack / OGNL

- a) ValueStack is a type of stack, it contains some objects or requests. ValueStack is used for hiding some data internally and execute in execute() method.

4. OGNL (Object Graph Navigation Language) is one type of Expression Language.

- a) OGNL is used to simplify the ActionContext contains data. These OGNL expressions are used to display the ValueStack object values in browser.
- b) Push the values in ValueStack object, then the values is mapped to JSP pages through property tag.

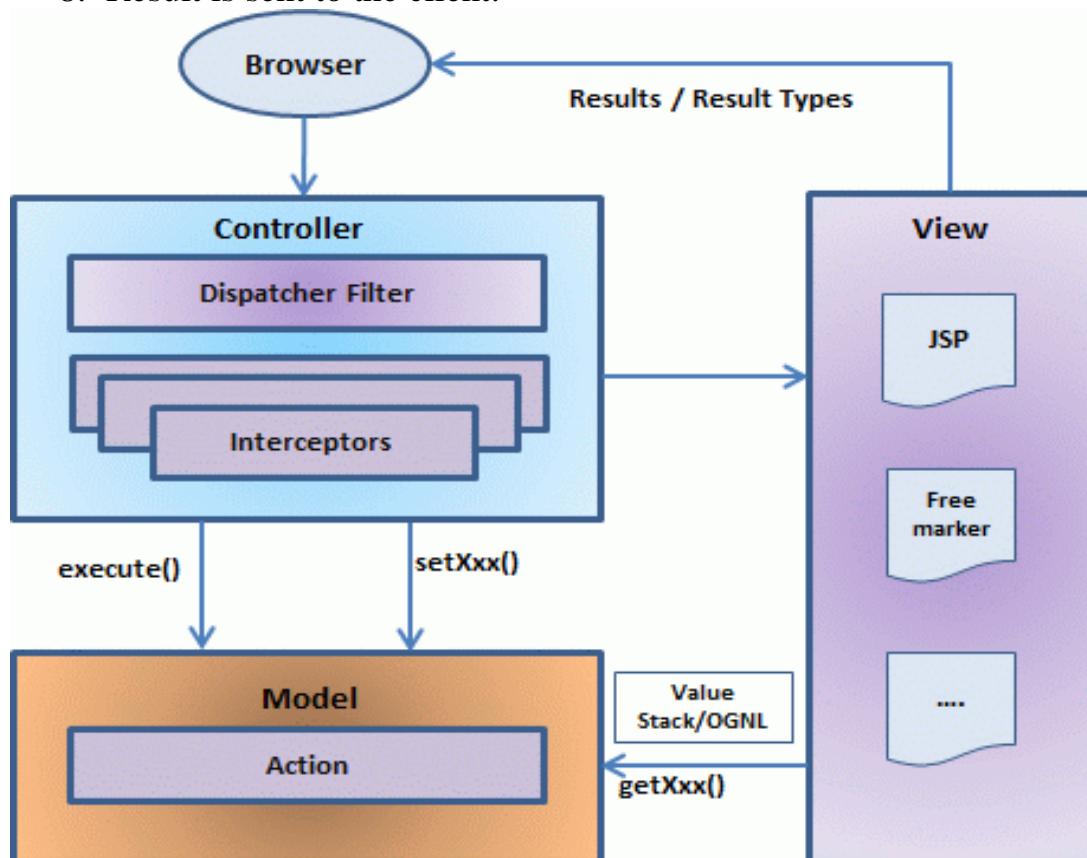
5. Result types

- a) Results are **UI** that represents the user requirement **from the application** onto the client browser



Life Cycle of Struts 2:

1. When a request comes web container **maps** the request in the **web.xml** and **calls the controller (FilterDispatcher)**.
2. **FilterDispatcher** **calls the** **ActionMapper** to find an Action to be invoked.
3. **FilterDispatcher** **calls the** **ActionProxy**.
4. **ActionProxy** **reads the** information of **action** and **interceptor stack** from configuration file using configuration manager (struts.xml) and **invoke the ActionInvocation**.
5. ActionInvocation **calls the** all interceptors one by one and then invoke the action to generate the result.
6. Action is executed ActionInvocation again calls the all interceptors in reverse order.
7. Control is returned to the FilterDispatcher.
8. Result is sent to the client.



Disadvantage of Struts 2:

1. **Bigger Learning Curve** – To use MVC with Struts, you have to be comfortable with the standard JSP, Servlet APIs and a large & elaborate framework.
2. **Poor Documentation** – Compared to the standard servlet and JSP APIs, Struts has fewer online resources, and many first-time users find the online Apache documentation confusing and poorly organized.
3. **Less Transparent** – With Struts applications, there is a lot more going on behind the scenes than with normal Java-based Web applications which makes it difficult to understand the framework.

Struts 2 MVC Design Pattern

- Action - model
- Result - view
- FilterDispatcher - controller

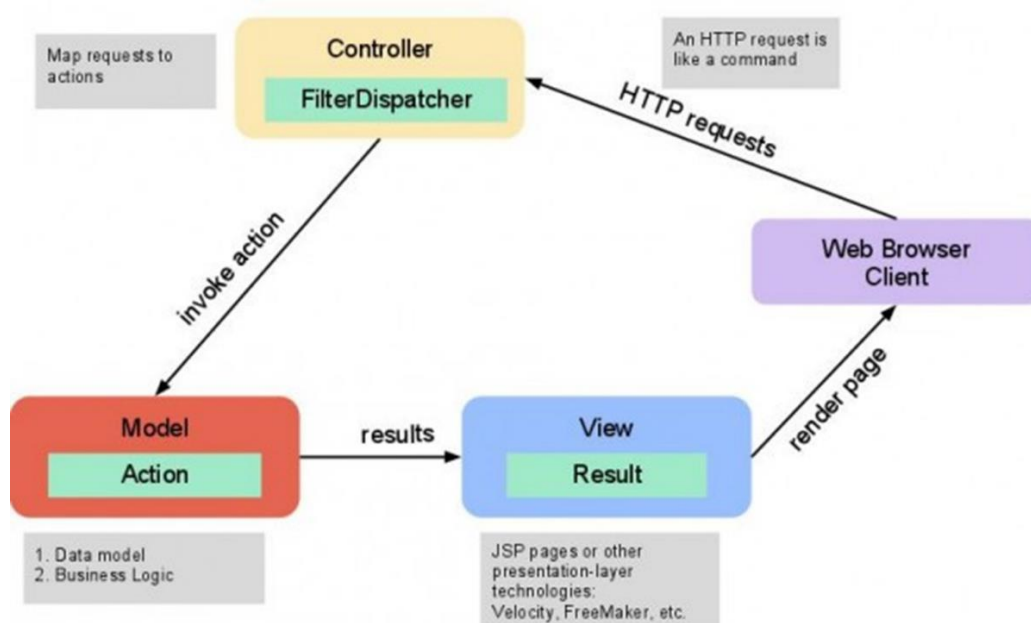
The role each module plays

Controller's job is to map incoming HTTP requests to actions. Those mapping are defined by using XML-based configuration(struts.xml) or Java annotations.

Model in Struts 2 is actions. Each action is defined and implemented by following the framework defined contract(e.g. consist an execute() method). Model component consists the data storage and business logic. Each action is an encapsulation of requests and is placed ValueStack.

View is the presentation component of MVC pattern. In spire of common JSP files, other techniques such as tilts, velocity, freemaker, etc. can be combined to provide a flexible presentation layer.

Interactions among each MVC module



Request Life Cycle:

- Initially User sends a request to the server for requesting for some resource (for example- pages).
- By looking at the request The Filter Dispatcher determines the appropriate Action.
- Validation, file upload etc functionalities are applied by Configured interceptor. For example- Selected action is performed based on the request operation
- Again, configured interceptors are applied to do any post-processing if required
- Finally, the result is prepared by the view and returns the result to the user

Example of Struts 2:**Create Action Class: (HelloWorldAction)**

```
package struts2;
```

```
public class HelloWorldAction {  
    private String name;  
  
    public String execute() throws Exception {  
        return "success";  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

.java file

Create a View: (HelloWorld.jsp)

```
<%@ page contentType = "text/html; charset = UTF-8" %>  
<%@ taglib prefix = "s" uri = "/struts-tags" %>  
  
<html>  
    <head>  
        <title>Hello World</title>  
    </head>
```



```
<body>
  Hello World, <s:property value = "name"/>
</body>
</html>
```

Create Main Page: (index.jsp)

```
<%@ page language = "java" contentType = "text/html; charset = ISO-8859-1"
  pageEncoding = "ISO-8859-1"%>
<%@ taglib prefix = "s" uri = "/struts-tags"%>
  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

```
<html>
  <head>
    <title>Hello World</title>
  </head>

  <body>
    <h1>Hello World From Struts2</h1>
    <form action = "hello">
      <label for = "name">Please enter your name</label><br/>
      <input type = "text" name = "name"/>
      <input type = "submit" value = "Say Hello"/>
    </form>
  </body>
</html>
```

Configuration Files (struts.xml)

```
<?xml version = "1.0" Encoding = "UTF-8"?>
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
  <constant name = "struts.devMode" value = "true" />

  <package name = "helloworld" extends = "struts-default">
    <action name = "hello">
```

```
class = "struts2.HelloWorldAction"
method = "execute">
<result name = "success">/HelloWorld.jsp</result>
</action>
</package>
</struts>
```

(web.xml)

```
<?xml version = "1.0" Encoding = "UTF-8"?>
<web-app xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xmlns = "http://java.sun.com/xml/ns/javaee"
  xmlns:web = "http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id = "WebApp_ID" version = "3.0">

  <display-name>Struts 2</display-name>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>
      org.apache.struts2.dispatcher.FilterDispatcher
    </filter-class>
  </filter>

  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

Actions:

In struts 2, action class is **POJO (Plain Old Java Object)**. POJO means you are not forced to implement any interface or extend any class. Generally, execute method should be specified that represents the business logic. The simple action class may look like:

```
Welcome.java
package javastruts;
public class Welcome {
    public String execute(){
        return "success";
    }
}
```

Action Interface - A convenient approach is to **implement** the **com.opensymphony.xwork2.Action** interface that defines **5 constants** and **one execute method**.

Action interface provides 5 constants that can be returned from the action class. They are:

1. **SUCCESS** indicates that action execution is successful and a success result should be shown to the user.
2. **ERROR** indicates that action execution is failed and a error result should be shown to the user.
3. **LOGIN** indicates that user is not logged-in and a login result should be shown to the user.
4. **INPUT** indicates that validation is failed and a input result should be shown to the user again.
5. **NONE** indicates that action execution is successful but no result should be shown to the user.

Let's see what values are assigned to these constants:

1. `public static final String SUCCESS = "success";`
2. `public static final String ERROR = "error";`
3. `public static final String LOGIN = "login";`
4. `public static final String INPUT = "input";`
5. `public static final String NONE = "none";`

Method of Action Interface

Action interface contains only **one method** `execute` that should be implemented overridden by the action class even if you are not forced.

`public String execute();`

Role of Action

1. Perform as a **model**
 - a. Action performs as a model by **encapsulating the actual work** to be done for a given request based on input parameters.
 - b. **Encapsulation** is done using the **execute()** method. The code inside this method should only hold the business logic to serve a Request
2. Serves as a **data Carrier**
 - a. Action serves as **data Carrier from Request** to the **View**. Action being the Model component of the framework carries the data around.
 - b. The data it requires is held locally which makes it easy to access using JavaBeans properties during the actual execution of the business logic
 - c. The **execute() method** simply references the data using JavaBean properties
3. Helps **Determine Result**
 - a. Action determines the Result that will render the View that will be returned in the requests response.
 - b. This is achieved by returning a control string that selects the result that should be rendered.
 - c. The value return string must be identical to the result name as configured.

Interceptors:

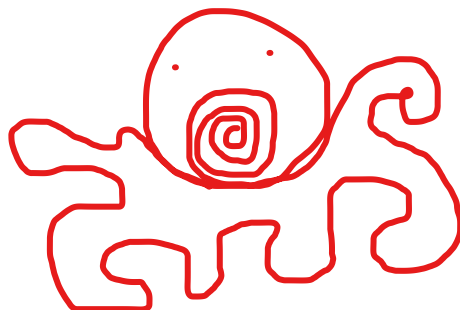
Interceptors are conceptually the same as **servlet filters** or the **JDKs Proxy class**. Interceptors allow for crosscutting functionality to be implemented separately from the action as well as the framework. You can achieve the following using interceptors –

- Providing **preprocessing** logic **before** the action is called.
- Providing **postprocessing** logic **after** the action is called.
- Catching exceptions so that alternate processing can be performed.

Many of the features provided in the Struts2 framework are implemented using interceptors;

Examples include exception handling, file uploading, lifecycle callbacks, etc. In fact, as Struts2 emphasizes much of its functionality on interceptors, it is not likely to have 7 or 8 interceptors assigned per action.

Struts 2 framework provides a good list of out-of-the-box interceptors that come preconfigured and ready to use.



Few of the important interceptors are listed below –

Sr. No.	Interceptors and Descriptions
1	alias Allows parameters to have different name aliases across requests.
2	checkbox Assists in managing check boxes by adding a parameter value of false for check boxes that are not checked.
3	conversionError Places error information from converting strings to parameter types into the action's field errors.
4	createSession Automatically creates an HTTP session if one does not already exist.
5	debugging Provides several different debugging screens to the developer.
6	execAndWait Sends the user to an intermediary waiting page while the action executes in the background.
7	exception Maps exceptions that are thrown from an action to a result, allowing automatic exception handling via redirection.
8	fileUpload Facilitates easy file uploading.
9	i18n Keeps track of the selected locale during a user's session.
10	logger Provides simple logging by outputting the name of the action being executed.
11	params Sets the request parameters on the action.
12	prepare This is typically used to do pre-processing work, such as setup database connections.
13	profile Allows simple profiling information to be logged for actions.
14	scope Stores and retrieves the action's state in the session or application scope.
15	ServletConfig Provides the action with access to various servlet-based information.
16	timer

	Provides simple profiling information in the form of how long the action takes to execute.
17	token Checks the action for a valid token to prevent duplicate form submission.
18	validation Provides validation support for actions

Results and Result Type

The `<results>` tag plays the role of a view in the Struts2 MVC framework. The action is responsible for executing the business logic. The next step after executing the business logic is to display the view using the `<results>` tag.

Often there is some navigation rules attached with the results. For example, if the action method is to authenticate a user, there are three possible outcomes.

- Successful Login
- Unsuccessful Login - Incorrect username or password
- Account Locked

In this scenario, the action method will be configured with three possible outcome strings and three different views to render the outcome.

Struts comes with a number of predefined result types and whatever we've already seen that was the default result type dispatcher, which is used to dispatch to JSP pages.

Struts allow you to use other markup languages for the view technology to present the results and popular choices include Velocity, Freemaker, XSLT and Tiles.

The Dispatcher Result Type

The dispatcher result type is the default type, and is used if no other result type is specified. It's used to forward to a servlet, JSP, HTML page, and so on, on the server. It uses the `RequestDispatcher.forward()` method.

We saw the "shorthand" version in our earlier examples, where we provided a JSP path as the body of the result tag.

```
<result name = "success">
    /HelloWorld.jsp
</result>
```

We can also specify the JSP file using a `<param name = "location">` tag within the `<result...>` element as follows –

```
<result name = "success" type = "dispatcher">
    <param name = "location">
```

```
    /HelloWorld.jsp  
  </param >  
</result>
```

We can also supply a `parse parameter`, which is `true` by default. The parse parameter determines whether or not the location parameter will be parsed for OGNL expressions.

The FreeMaker Result Type

In this example, we are going to see how we can use FreeMaker as the view technology. Freemake is a popular templating engine that is used to generate output using predefined templates. Let us now create a Freemake template file called `hello.fm` with the following contents –

```
Hello World ${name}
```

The above file is a template where name is a parameter which will be passed from outside using the defined action. You will keep this file in your CLASSPATH. Next, let us modify the struts.xml to specify the result as follows –

```
<action name = "hello"  
  class = "packageName.HelloWorldAction"  
  method = "execute">  
  <result name = "success" type = "freemarker">  
    <param name = "location">/hello.fm</param>  
  </result>  
</action>
```

The Redirect Result Type

The redirect result type calls the standard `response.sendRedirect()` method, causing the browser to create a new request to the given location.

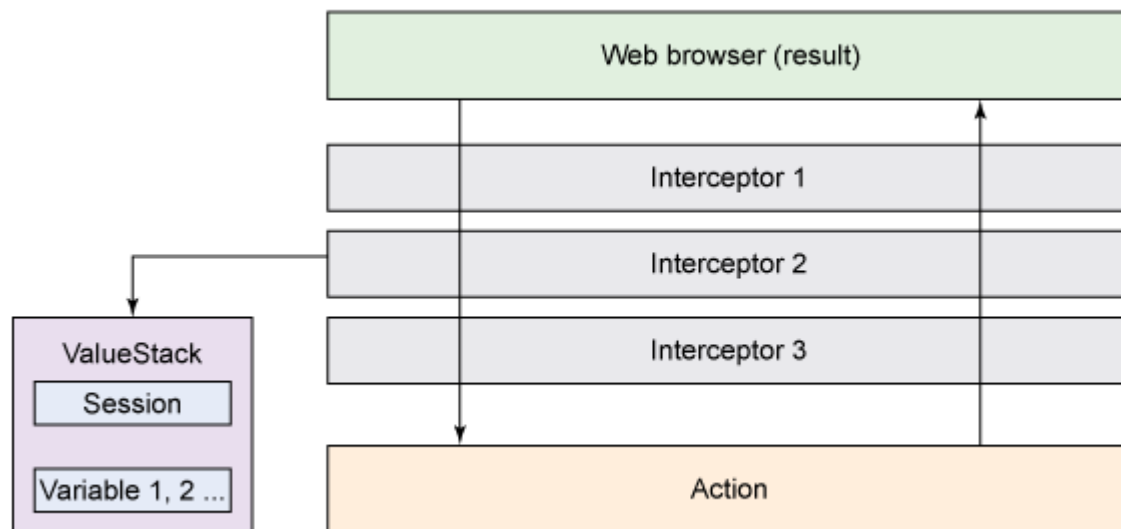
We can provide the location either in the body of the `<result...>` element or as a `<param name = "location">` element. Redirect also supports the parse parameter. Here's an example configured using XML –

```
<action name = "hello"  
  class = "com.tutorialspoint.struts2.HelloWorldAction"  
  method = "execute">  
  <result name = "success" type = "redirect">  
    <param name = "location">  
      /NewWorld.jsp  
    </param >  
  </result>  
</action>
```

Value Stack

A Value Stack is simply a stack that contains application specific objects such as action objects and other model object. At the execution time, action is placed on the top of the stack. We can put objects in the Value Stack, query it and delete it. Before an Action execute() method is called for business processing, framework moves all the data to the Value Stack for request processing.

Framework manipulates the Value Stack data during action execution, render the result and send the response page.

**Value Stack Object:****1. Temporary Object:**

Temporary objects are created during the execution of a page. Example- the current iteration value, looped over JSP tag etc.

2. Model Object:

If Model object is used in struts application, the current model object is placed on the value stack before the action execution

3. The Action Object:

It is the current action object being executed.

4. Named Object:

It includes application, session, request, attributes and parameters and refers to the corresponding servlet scopes.

Methods of Value Stack interface:

There are many methods in Value Stack interface. The commonly used methods are as follows:

1. `public String findString(String expr)` finds the string by evaluating the given expression.
2. `public Object findValue(String expr)` finds the value by evaluating the specified expression.
3. `public Object findValue(String expr, Class c)` finds the value by evaluating the specified expression.
4. `public Object peek()` It returns the object located on the top of the stack.
5. `public Object pop()` It returns the object located on the top of the stack and removes it.
6. `public void push(Object o)` It puts the object on the top of the stack.
7. `public void set(String key, Object value)` It sets the object on the stack with the given key. It can be get by calling the `findValue(key)` method.
8. `public int size()` It returns the number of objects from the stack.

OGNL (Object Graph Navigation Language):

The Object-Graph Navigation Language (OGNL) is a powerful expression language that is used to reference and manipulate data on the ValueStack. OGNL also helps in data transfer and type conversion.

The OGNL is very similar to the JSP Expression Language. OGNL is based on the idea of having a root or default object within the context. The properties of the default or root object can be referenced using the markup notation, which is the pound symbol.

OGNL Supports:

1. Type Conversion
2. Calling method
3. Collection Manipulation
4. Expression evaluation

As mentioned earlier, OGNL is based on a context and Struts builds an ActionContext map for use with OGNL. The ActionContext map consists of the following –

1. **Application** – Application scoped variables
2. **Session** – Session scoped variables
3. **Root / value stack** – All your action variables are stored here
4. **Request** – Request scoped variables
5. **Parameters** – Request parameters
6. **Attributes** – The attributes stored in page, request, session and application scope

OGNL Execution flow

- Struts framework receives the client request name and value in the form of string
- OGNL expression scan the value stack to locate the destination property where value need to be saved
- OGNL performs the type conversion of the request data and move the value to the property by invoking the corresponding Setter method
- Results access the value stack using OGNL expression tags to render the response.
- While rendering the view, value is converted from java type to string type.
- The objects in the value stack can be directly referenced. If 'name' is a property of an action class, it can be referenced as–

<s:property value = "name"/>

If you have an attribute in session called "login" you can retrieve it as –

<s:property value = "#session.login"/>

OGNL also supports dealing collections known as Map, List and Set. Example to display a dropdown list of colors, you could do –

<s:select name = "color" list = "{ 'red','yellow','green'}" />

The OGNL expression is clever to interpret the "red","yellow","green" as colours and build a list based on that.

JSON

JavaScript Object Notation is an open-standard file format or data interchange format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value). It is a very common data format, with a diverse range of applications, such as serving as replacement for XML in AJAX systems.

JSON is a language-independent data format. It was derived from JavaScript, but many modern programming languages include code to generate and parse JSON-format data. JSON filenames use the extension .json

Exchanging data:

Data exchange between a browser and a server, is always in text format. We can convert any JavaScript object into JSON and send JSON to the server.

We can also convert any JSON received from the server into JavaScript object. This how simply we can work with the data as JavaScript object, with no complications.

1. **Sending Data:**

We can convert the JavaScript object into JSON and send it to server:

Example- `var data = {"name": "Geeta", "age": 31, "city": "Banglore"};`

2. Receiving Data:

If we receive data in JSON format we can convert it into JavaScript object:

Example-

```
var data = {"name": "Geeta", "age": 31, "city": "Banglore"};
var obj = JSON.parse(data);
document.getElementById("MyDemo").innerHTML= obj.name;
```

Why use JSON:

1. Since JSON format is text only it is very easy to use for data transfer, and can be used as a data format by any programming language
2. JSON.parse() parse a JSON string, constructing the JavaScript value or object described by the string. So if we receive data from a server, in JSON format, we can use it like any other JavaScript
3. The basic use of JSON is transfer data between a server and web application
4. JSON is used with modern programming language Web services and API's uses JSON format to provide public data
5. Majorly JSON is used for JavaScript based applications that include browser extensions and websites
6. JSON format is used for serializing and transmitting structured data over network connection.

JSON Syntax:

A JSON document may contain information separated by following separators/token.

- ":" to separate name from value
- "," to separate name-value pairs
- "{" and "}" for objects
- "[" and "]" for arrays

JSON name-value pairs example

Name-value pairs have a colon between them as in "name" : "value".

JSON names are on the left side of the colon. They need to be wrapped in double quotation marks, as in "name", and can be any valid string. Within each object, keys need to be unique.

Each name-value pair is separated by a comma, so the JSON looks like this:

"name" : "value", "name" : "value", "name": "value" e.g.



```
{  
  "color" : "Purple",  
  "id" : "210"  
}
```

JSON Data Type:

JSON's basic data types are:

1. **Number:** a signed decimal number that may contain a fractional part and may use exponential E notation, but cannot include non-numbers such as NaN. The format makes no distinction between integer and floating-point. JavaScript uses a double-precision floating-point format for all its numeric values, but other languages implementing JSON may encode numbers differently.
2. **String:** a sequence of zero or more Unicode characters. Strings are delimited with double-quotation marks and support a backslash escaping syntax.
3. **Boolean:** either of the values true or false
4. **Array:** an ordered list of zero or more values, each of which may be of any type. Arrays use square bracket notation with comma-separated elements.
5. **Object:** an unordered collection of name–value pairs where the names (also called keys) are strings. Objects are intended to represent associative arrays, where each key is unique within an object. Objects are delimited with curly brackets and use commas to separate each pair, while within each pair the colon ':' character separates the key or name from its value.
6. **null:** an empty value, using the word null

Example:

```
{  
  "firstName": "John",  
  "lastName": "Smith",  
  "isAlive": true,  
  "age": 27,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": "10021-3100"  
  },  
  "phoneNumbers": [  
    {  
      "type": "home",
```

```
"number": "212 555-1234"
}, {
  "type": "office",
  "number": "646 555-4567"
},{
  "type": "mobile",
  "number": "123 456-7890"
}
],
"children": [],
"spouse": null
}
```

JSON comparison with XML:

Both JSON and XML are "self describing" (human readable), hierarchical (values within values), can be parsed and used by lots of programming languages and XML can be fetched with an XMLHttpRequest. JSON is unlike XML because it doesn't use end tag, is shorter, quicker to read and write, and can use arrays.

The biggest difference is:

XML has to be parsed with an XML parser. JSON can be parsed by a standard JavaScript function.

JSON Example

```
{ "employees": [
  { "firstName": "John", "lastName": "Doe" },
  { "firstName": "Anna", "lastName": "Smith" },
  { "firstName": "Peter", "lastName": "Jones" }
]}
```

XML Example

```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```


Differentiate between JSON and XML

JSON	XML
JSON object has a type	XML data is typeless
JSON types: string, number, array, Boolean	All XML data should be string
Data is readily accessible as JSON objects	XML data needs to be parsed.
JSON is supported by most browsers.	Cross-browser XML parsing can be tricky
JSON has no display capabilities.	XML offers the capability to display data because it is a markup language.
JSON supports only text and number data type.	XML support various data types such as number, text, images, charts, graphs, etc. It also provides options for transferring the structure or format of the data with actual data.
Retrieving value is easy	Retrieving value is difficult
Supported by many Ajax toolkit	Not fully supported by Ajax toolkit
A fully automated way of deserializing/serializing JavaScript.	Developers have to write JavaScript code to serialize/de-serialize from XML
Native support for object.	The object has to be express by conventions - mostly missed use of attributes and elements.
It supports only UTF-8 encoding.	It supports various encoding.
It doesn't support comments.	It supports comments.
JSON files are easy to read as compared to XML.	XML documents are relatively more difficult to read and interpret.
It does not provide any support for namespaces.	It supports namespaces.
It is less secured.	It is more secure than JSON.

JSON with Java:


JSON (JavaScript Object Notation) is a lightweight, text-based, language-independent data exchange format that is easy for humans and machines to read and write.

The Java API for JSON Processing **JSON.simple** is a simple Java library that allow parse, generate, transform, and query JSON.

How to set up Environment:

Before you start with encoding and decoding JSON using Java, you need to install any of the JSON modules available. For this tutorial we have downloaded and installed JSON.simple and have added the location of `json-simple-1.1.1.jar` file to the environment variable CLASSPATH.

The org.json.simple package contains important classes for JSON API

- JSONValue
 - JSON Object
 - JSONArray
 - JsonString
 - JsonNumber
- 

Mapping between JSON and Java entities

JSON.simple maps entities from the left side to the right side while decoding or parsing, and maps entities from the right to the left while encoding.

JSON	Java
string	java.lang.String
number	java.lang.Number
true false	java.lang.Boolean
null	null
array	java.util.List
object	java.util.Map

On decoding, the default concrete class of `java.util.List` is `org.json.simple.JSONArray` and the default concrete class of `java.util.Map` is `org.json.simple.JSONObject`.

Example:

Encoding JSON in Java:

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.LinkedHashMap;
import java.util.Map;
import org.json.simple.JSONArray;
import org.json.simple.JSONObject;
```

```
public class JSONWriteExample
{
    public static void main(String[] args) throws FileNotFoundException
    {
        // creating JSONObject
        JSONObject jo = new JSONObject();

        → // putting data to JSONObject
        jo.put("firstName", "John");
        jo.put("lastName", "Smith");
        jo.put("age", 25);

        // for address data, first create LinkedHashMap
        Map m = new LinkedHashMap(4);
        m.put("streetAddress", "21 2nd Street");
        m.put("city", "New York");
        m.put("state", "NY");
        m.put("postalCode", 10021);

        → // putting address to JSONObject
        jo.put("address", m);

        // for phone numbers, first create JSONArray
        JSONArray ja = new JSONArray();

        m = new LinkedHashMap(2);
        m.put("type", "home");
        m.put("number", "212 555-1234");

        // adding map to list
        ja.add(m);

        m = new LinkedHashMap(2);
        m.put("type", "fax");
        m.put("number", "212 555-1234");

        // adding map to list
        ja.add(m);

        // putting phoneNumbers to JSONObject
        → jo.put("phoneNumbers", ja);
```

key - value

```
// writing JSON to file:"JSONExample.json" in cwd
PrintWriter pw = new PrintWriter("JSONExample.json");
pw.write(jo.toJSONString());

pw.flush();
pw.close();
}
}
```

Output from file “JSONExample.json”:

```
{
  "lastName":"Smith",
  "address":{
    "streetAddress":"21 2nd Street",
    "city":"New York",
    "state":"NY",
    "postalCode":10021
  },
  "age":25,
  "phoneNumbers":[
    {
      "type":"home", "number":"212 555-1234"
    },
    {
      "type":"fax", "number":"212 555-1234"
    }
  ],
  "firstName":"John"
}
```

Decoding JSON in Java:

```
import java.io.FileReader;
import java.util.Iterator;
import java.util.Map;
import org.json.simple.JSONArray;
import org.json.simple.JSONObject;
import org.json.simple.parser.*;

public class JSONReadExample
```

```
{
    public static void main(String[] args) throws Exception
    {
        // parsing file "JSONExample.json"
        Object obj = new JSONParser().parse(new
        FileReader("JSONExample.json"));

        // typecasting obj to JSONObject
        JSONObject jo = (JSONObject) obj;


        // getting firstName and lastName
        String firstName = (String) jo.get("firstName");
        String lastName = (String) jo.get("lastName");

        System.out.println(firstName);
        System.out.println(lastName);

        // getting age
        long age = (long) jo.get("age");
        System.out.println(age);

        // getting address
        Map address = ((Map)jo.get("address"));

        // iterating address Map
        Iterator<Map.Entry> itr1 = address.entrySet().iterator();
        while (itr1.hasNext()) {
            Map.Entry pair = itr1.next();
            System.out.println(pair.getKey() + " : " + pair.getValue());
        }

         // getting phoneNumbers
        JSONArray ja = (JSONArray) jo.get("phoneNumbers");

        // iterating phoneNumbers
        Iterator itr2 = ja.iterator();

        while (itr2.hasNext())
        {
            itr1 = ((Map) itr2.next()).entrySet().iterator();
            while (itr1.hasNext()) {
                Map.Entry pair = itr1.next();
                System.out.println(pair.getKey() + " : " + pair.getValue());
            }
        }
    }
}
```

```
}  
}  
}  
}
```

Output :

```
John  
Smith  
25  
streetAddress : 21 2nd Street  
postalCode : 10021  
state : NY  
city : New York  
number : 212 555-1234  
type : home  
number : 212 555-1234  
type : fax
```