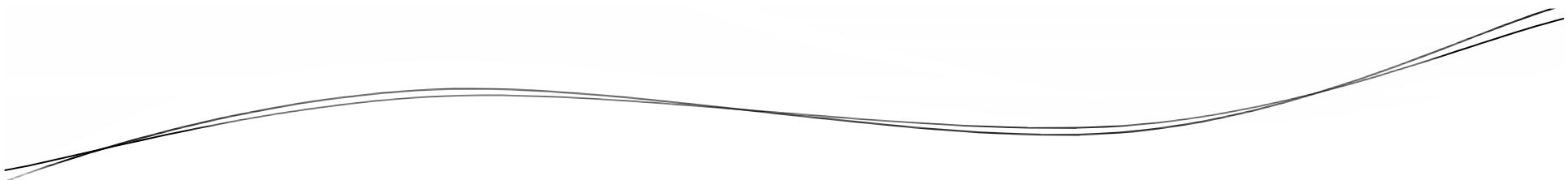
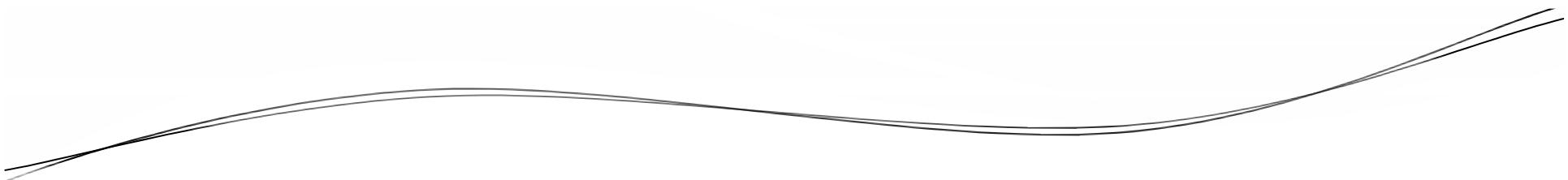


Java EE

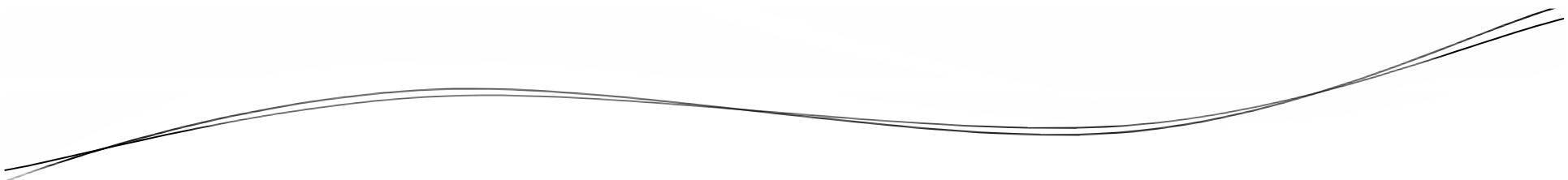


Objectives

- What is Java EE
- Java EE Components
- Need for Java EE
- Java EE Environment

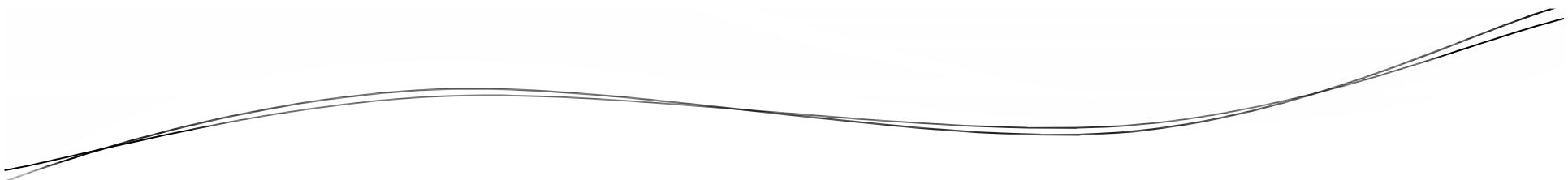


Java EE

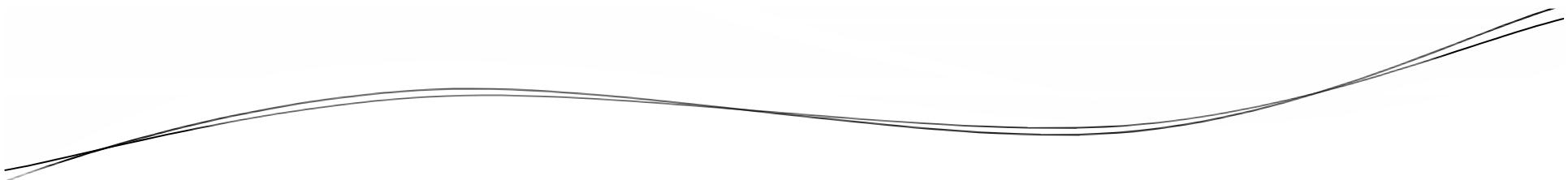


Java EE

- Java Enterprise Edition is a platform designed to create web based as well as enterprise level applications.
- Java EE emphasizes upon Component Driven Architecture.



Why Java EE

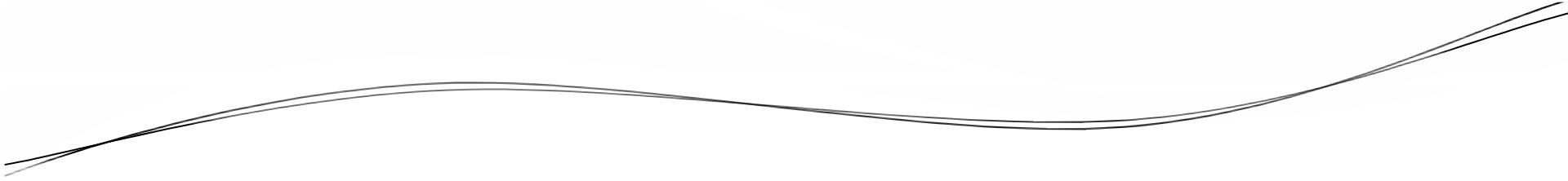


Why Java EE

- Would my application be able to support a set of thousand or more users?
- Would the service be available for 24x7x365?
- Would the response time suffer if user load increases?

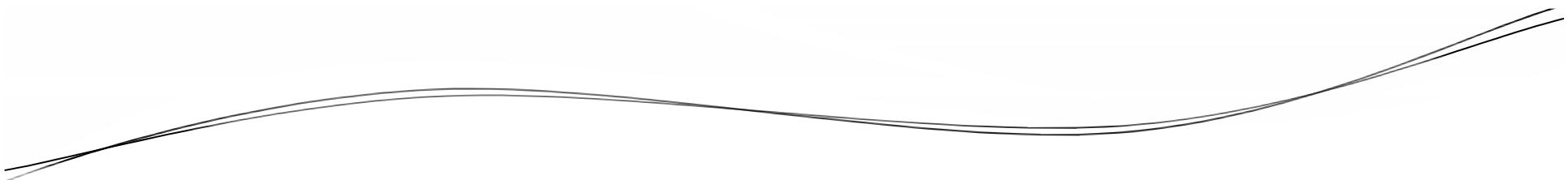
Why Java EE

- Would the hardware resource requirements increase day by day with increasing user load?
- Would my application be able to get integrated with external applications smoothly?
- Will anybody be able to hack, steal or corrupt my confidential data?

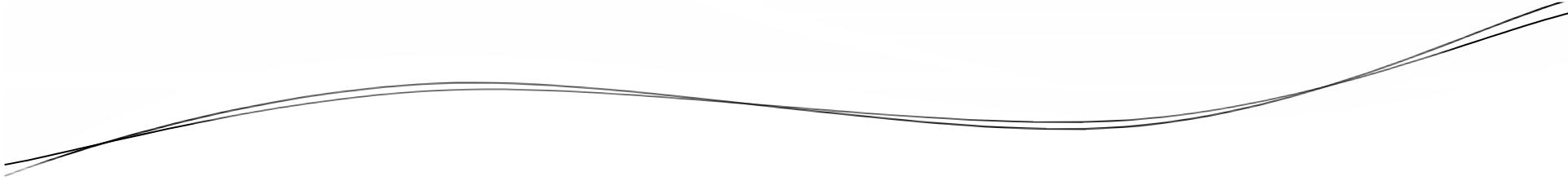


Quality of Service Requirements

- Scalability
- Availability
- Performance
- Flexibility
- Security
- Re-usability
- Asynchronous Messaging

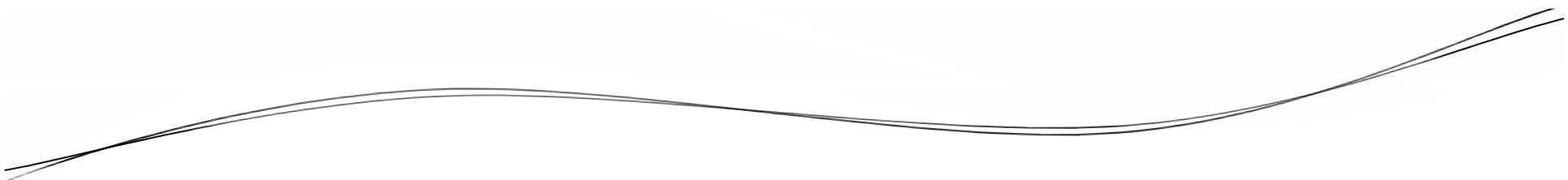


Components



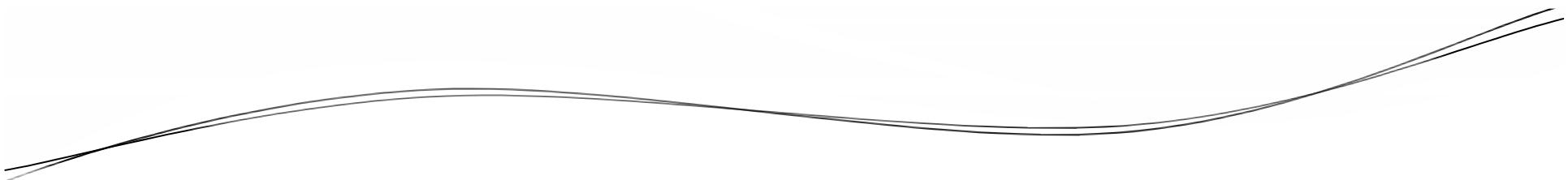
Components

- A component is an application level reusable unit.
- Components are divided into 2 types:
 - Unmanaged
 - Managed



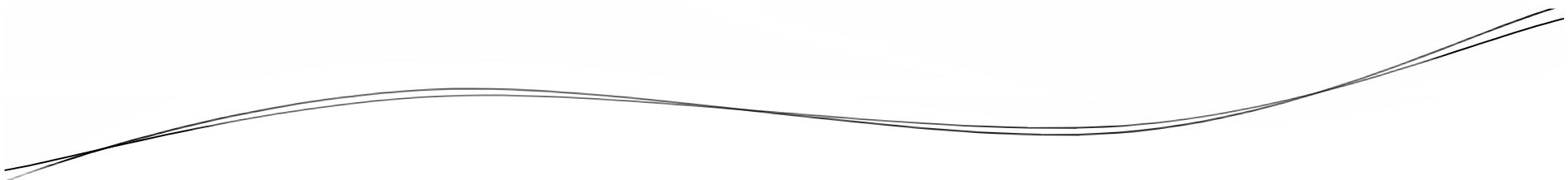
Components

- A component which is to be instantiated explicitly is called as an unmanaged component.



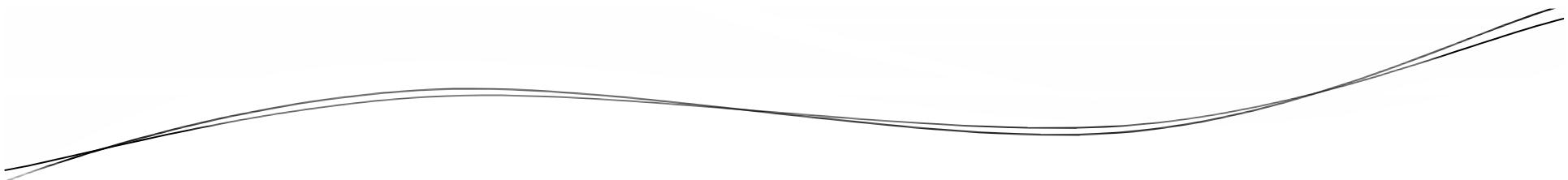
Components

- A component which gets instantiated implicitly is called as a managed component.
- Managed components are taken care by an environment known as Container.

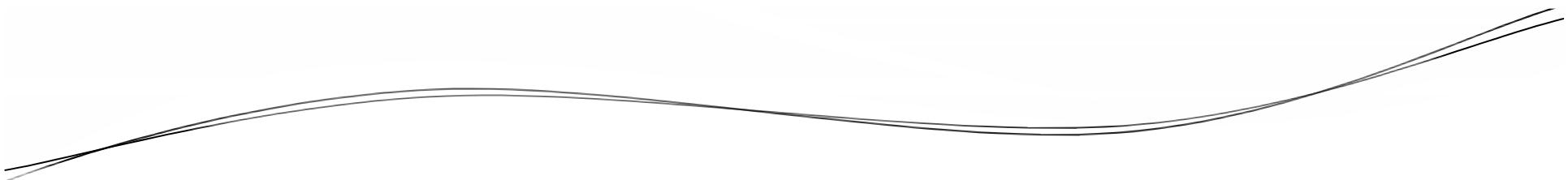


Components

- In Java EE, components are divided into 2 types:
 - Web Components
 - Business Components

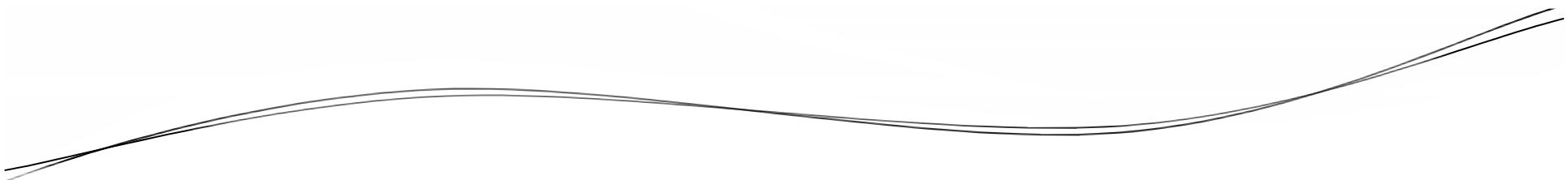


Web Components

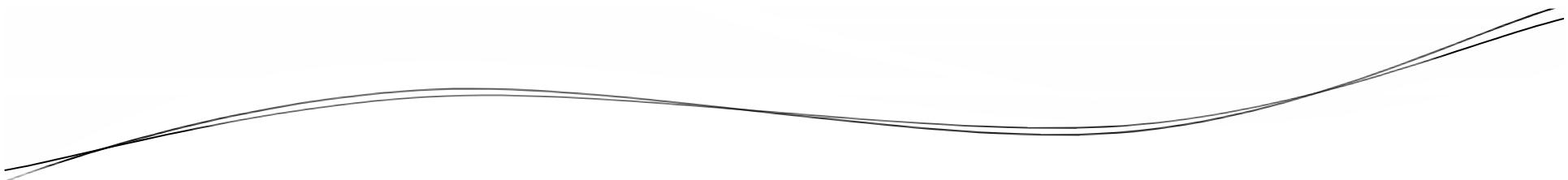


Web Components

- A component that is responsible for accepting a web request and generating a web response is called as a Web Component, e.g, Servlet and JSP.

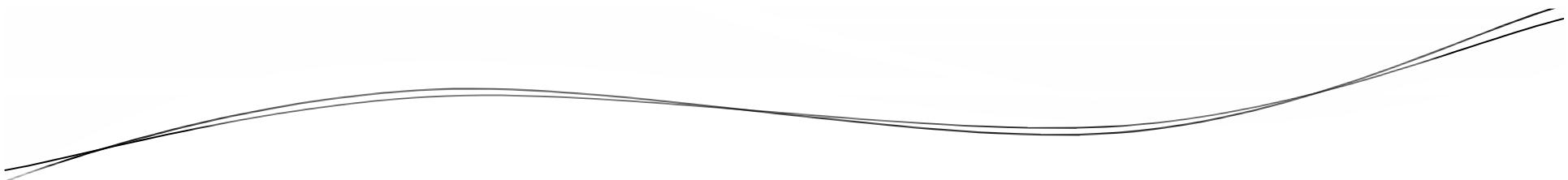


Business Components

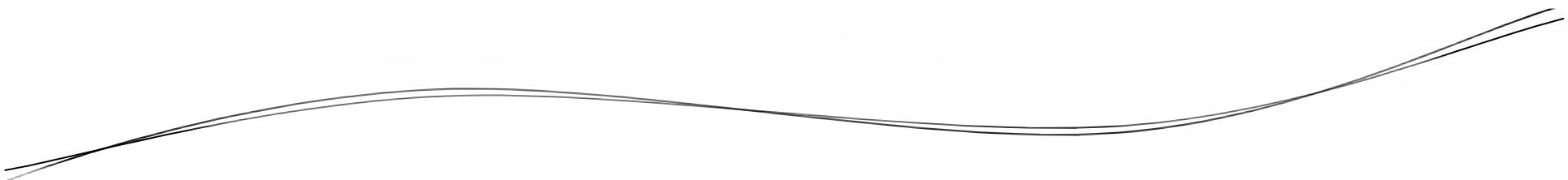


Business Components

- A component that is responsible for handling a business logic of the application is called as a Business Component, e.g, EJB.

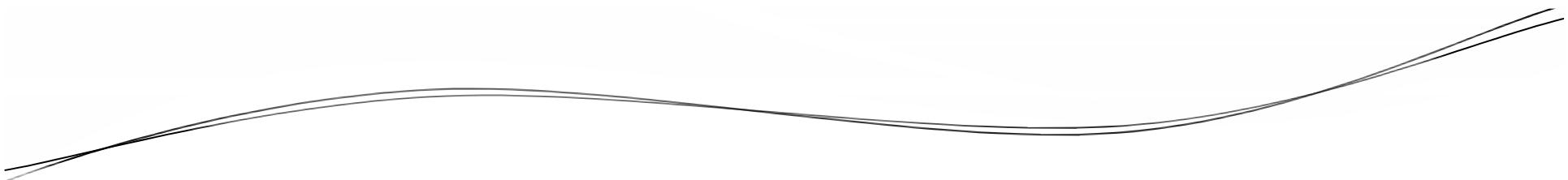


Container



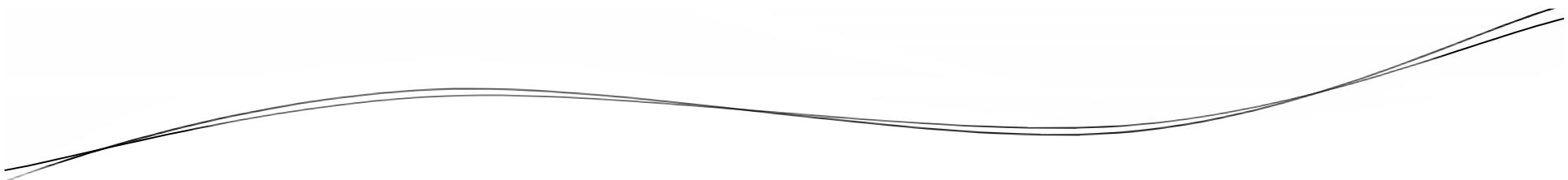
Container

- A container is a runtime environment responsible for managing the life cycle of a Java EE components.

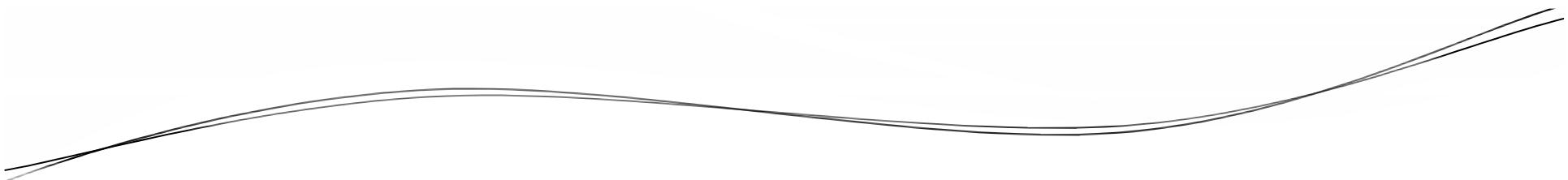


Container

- Containers are of 2 types:
 - Web Container
 - EJB Container

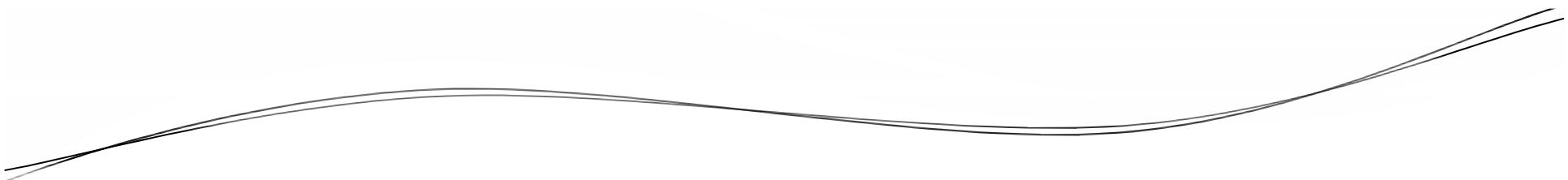


Web Container



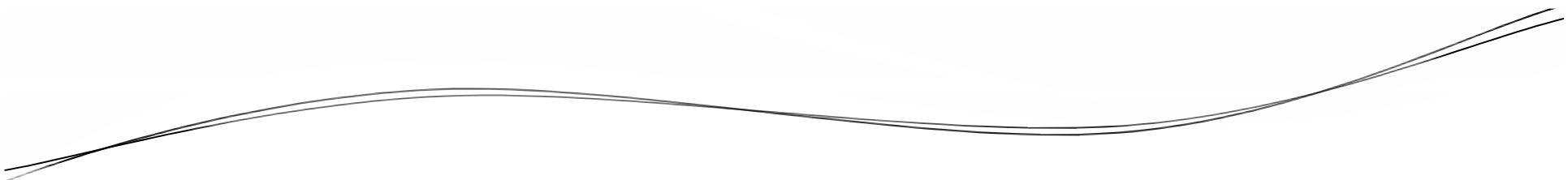
Web Container

- A Web Container is a runtime environment responsible for managing the life cycle of web components: Servlet and JSP.

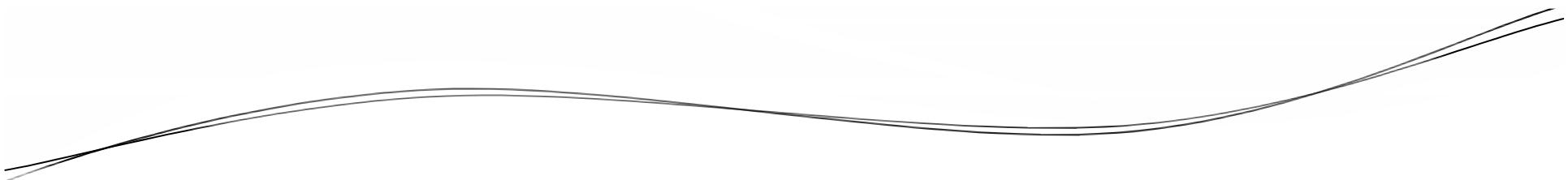


Web Container

- A web container is made available by a 3rd party product called as Web Server.
- The most commonly used web server is Apache Tomcat.

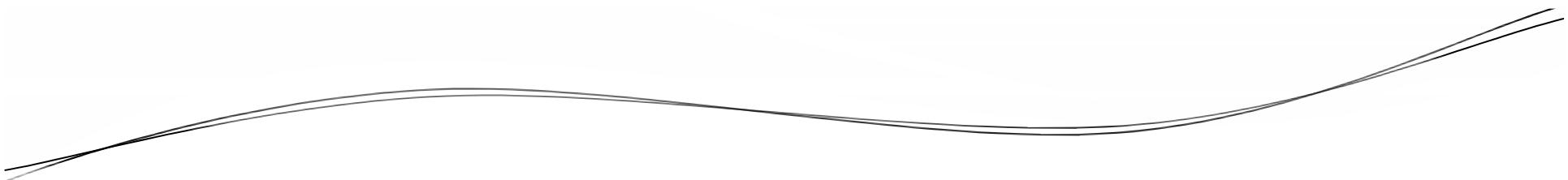


EJB Container



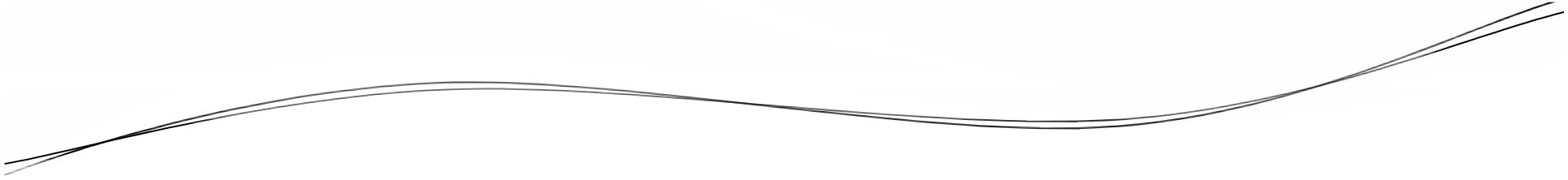
EJB Container

- An EJB Container is a runtime environment responsible for managing the life cycle of a business component: EJB.



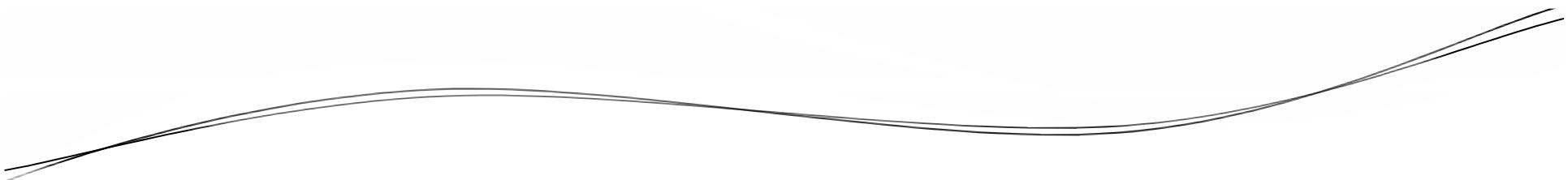
EJB Container

- An EJB Container is made available by a 3rd party product called as an Application Server.
- An Application Server is an extension to Web Server.



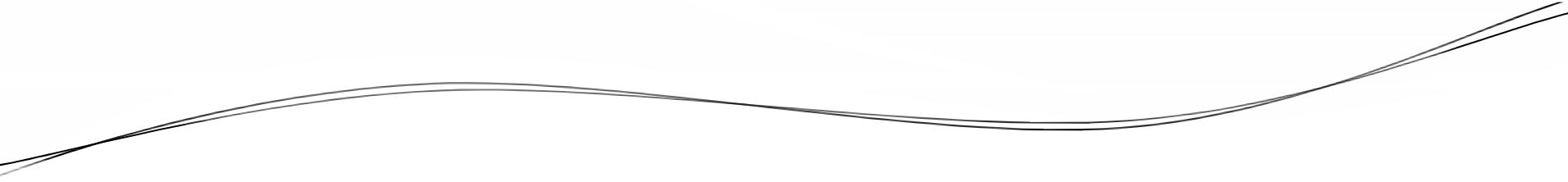
EJB Container

- There are different types of Application Servers:
 - Oracle Weblogic
 - IBM WebSphere
 - RedHat Jboss
 - Oracle Glassfish



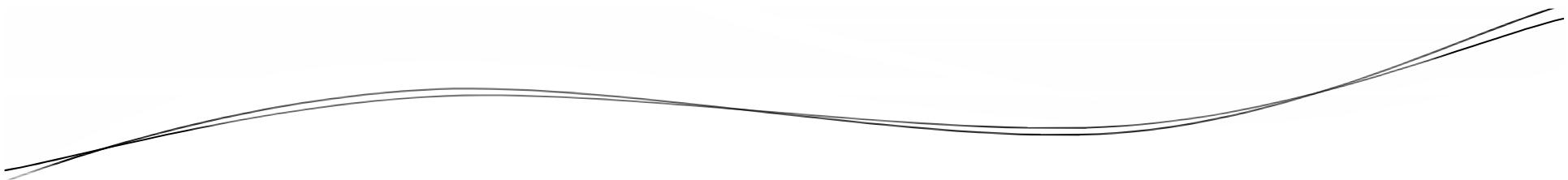
Let's Summarize

- What is Java EE
- Why Java EE
- Java EE Components
- Web Server and Application Server



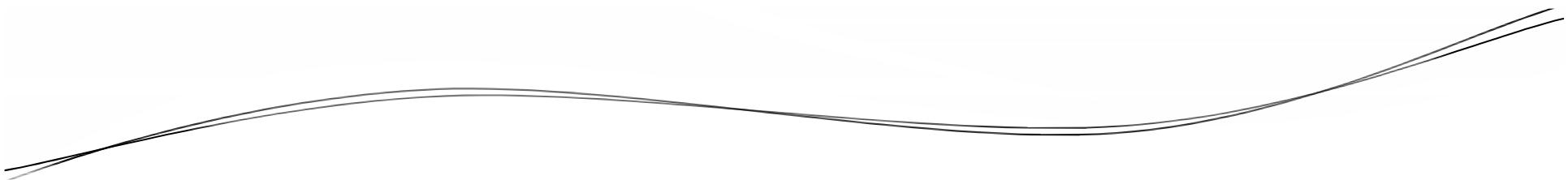
JDBC

By Rahul Barve



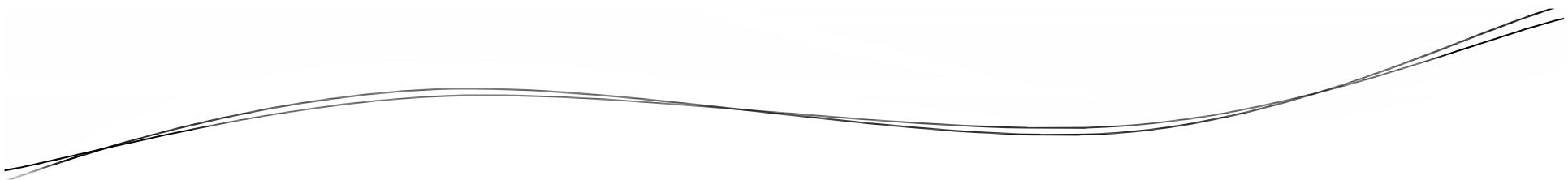
Objectives

- Introduction to JDBC
- Why JDBC
- JDBC Drivers
- JDBC Core API
- Executing Simple Queries
- Executing Parameterized Queries
- Transaction Management



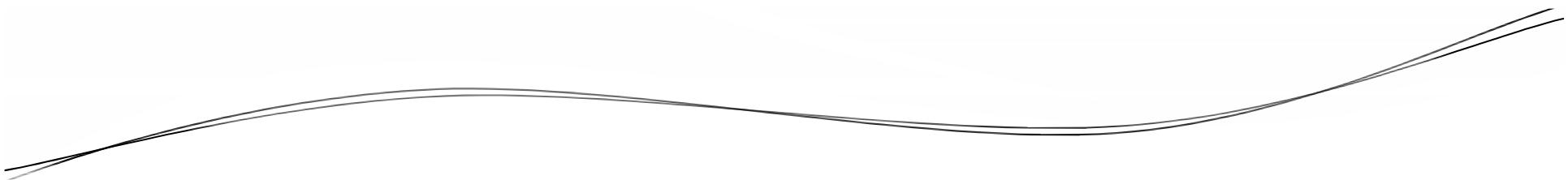
Introduction to JDBC

By Rahul Barve



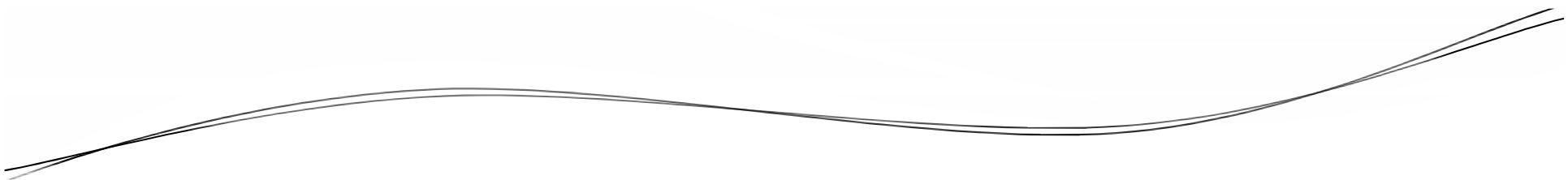
Introduction to JDBC

- JDBC stands for Java to Database Connectivity.
- It is an API that allows Java applications to interact with Database.



Why JDBC

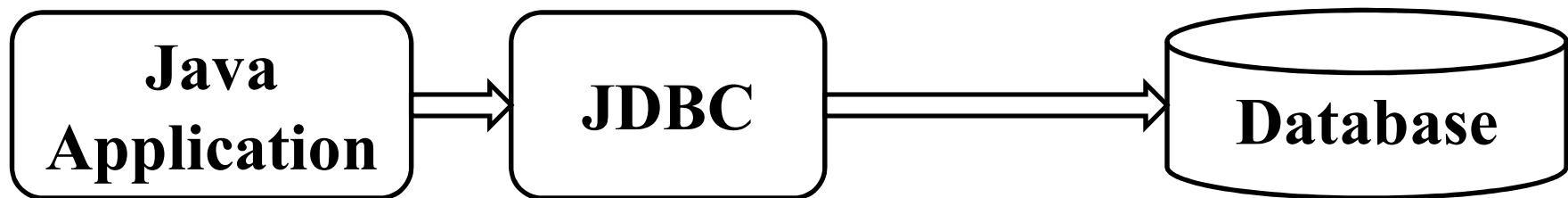
By Rahul Barve

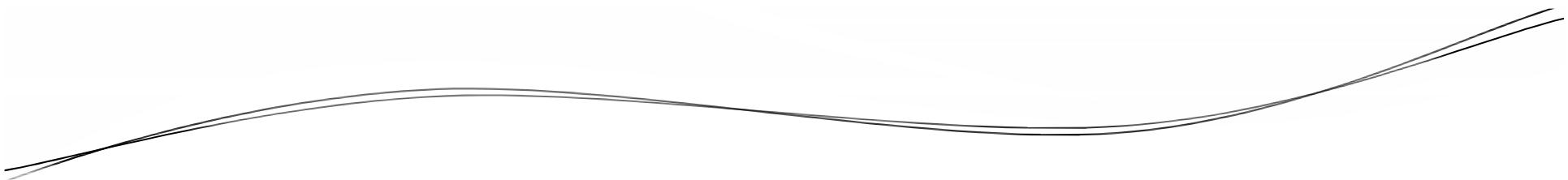


Why JDBC

- Applications need to write data into or load data from database.
- JDBC provides a channel to bridge the gap between a Java application and a Database.

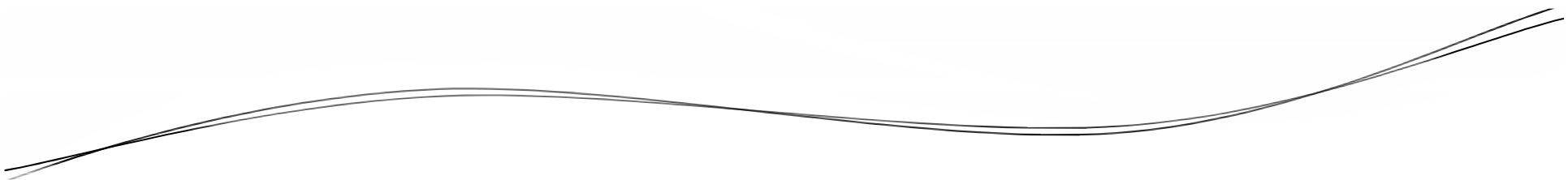
Why JDBC





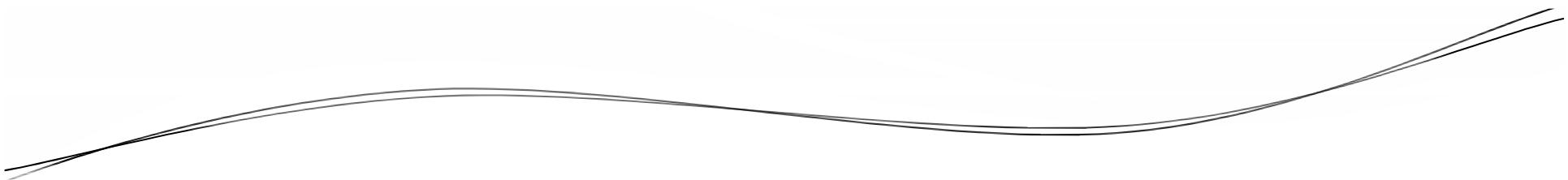
JDBC Driver

By Rahul Barve



JDBC Driver

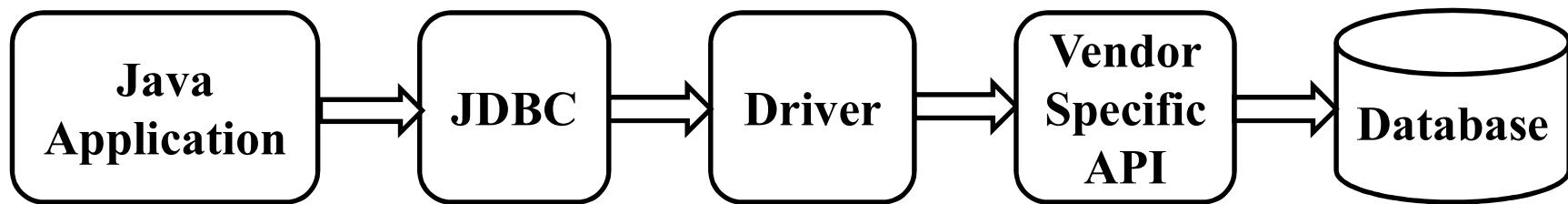
- Every DB vendor provides its own API that simplifies access for the client programs to connecting to the database.
- Such an API is known as a Vendor Specific API.

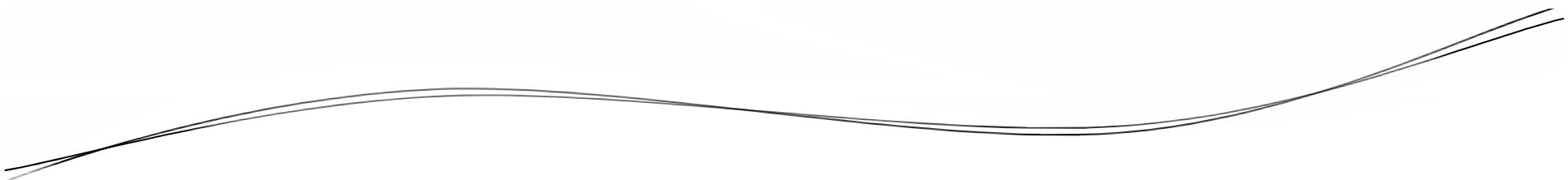


JDBC Driver

- Since both the APIs are written as per the proprietary standards, they are not compatible with each other.
- This mismatch is resolved by a mediator known as a Driver.

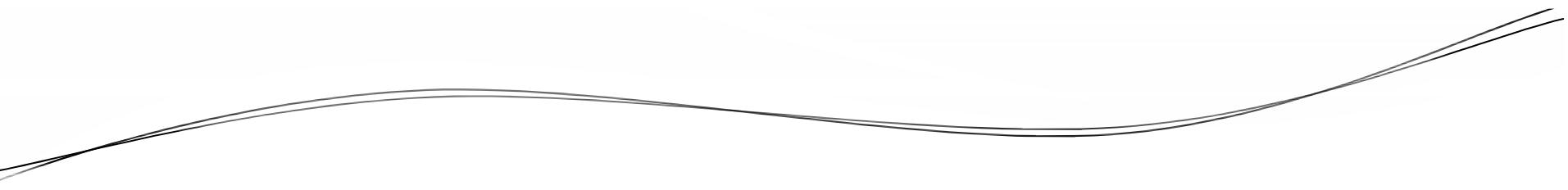
JDBC Driver





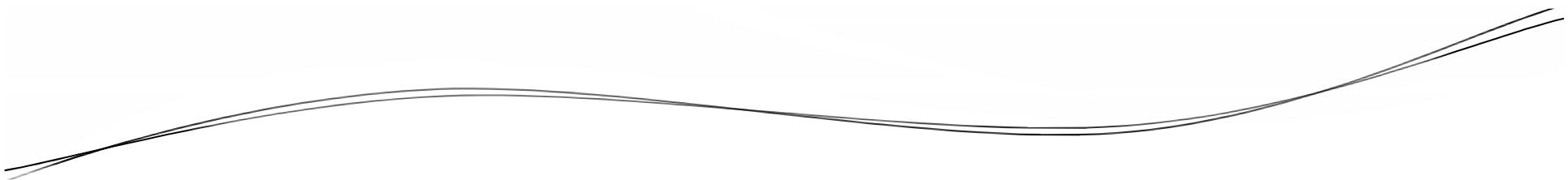
JDBC Driver

- There are 4 types of JDBC drivers:
 - Type 1
 - Type 2
 - Type 3
 - Type 4



Type 1 Driver

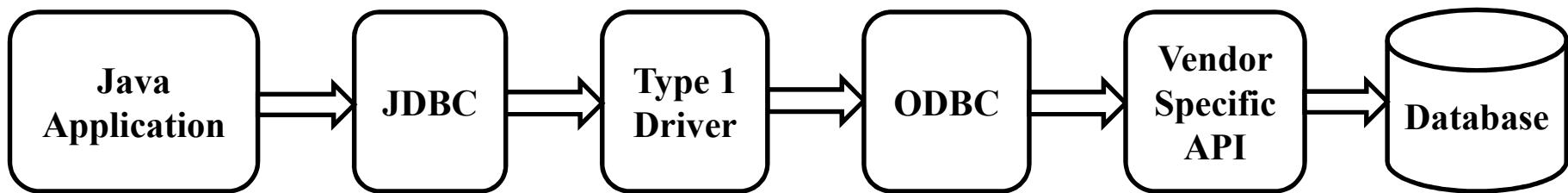
By Rahul Barve

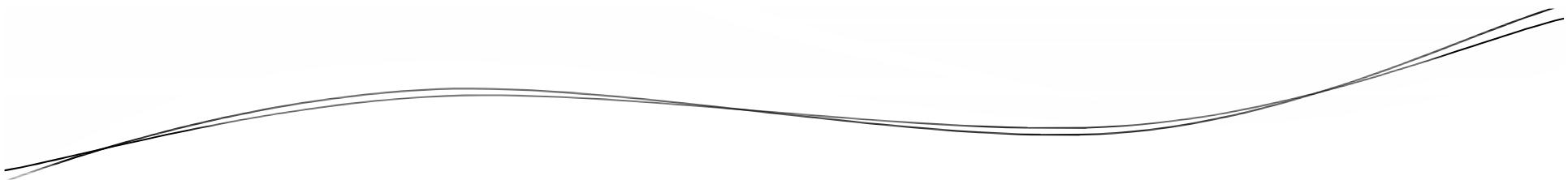


Type 1 Driver

- It is called as a JDBC – ODBC Bridge.
- It uses a 3rd party library known as ODBC which is provided by Microsoft.

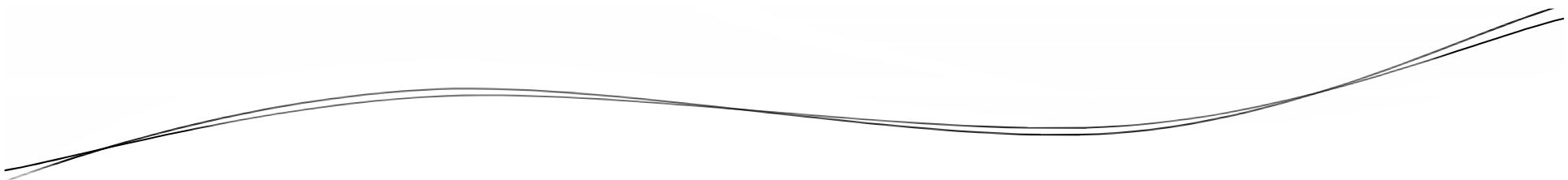
Type 1 Driver





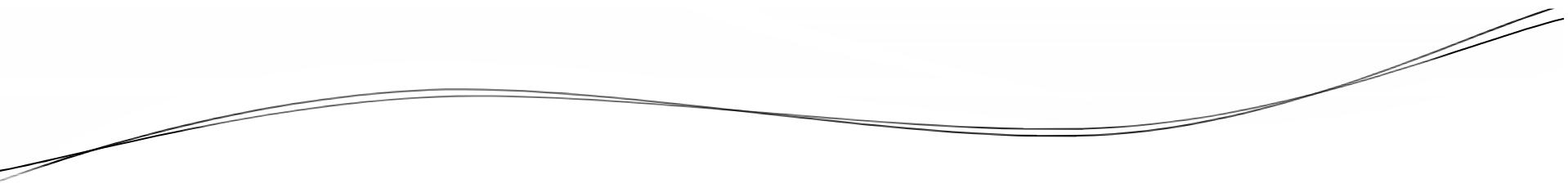
Type 1 Driver

- It is platform dependent.
- It is the slowest, takes much time for processing.
- Every client machine must have ODBC configuration setup.



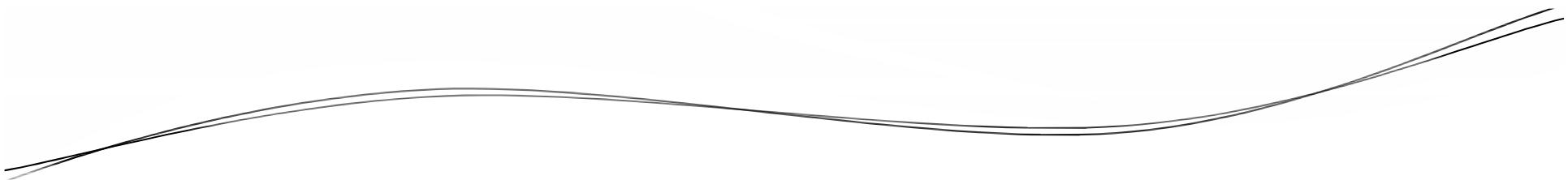
Type 1 Driver

- Suitable for simple desktop applications or even just for testing purpose.
- Not much recommended in case of large scale applications or even in production environment.



Type 2 Driver

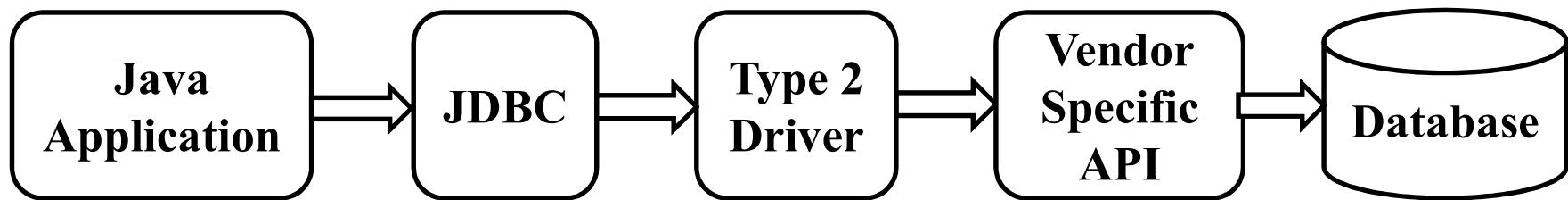
By Rahul Barve

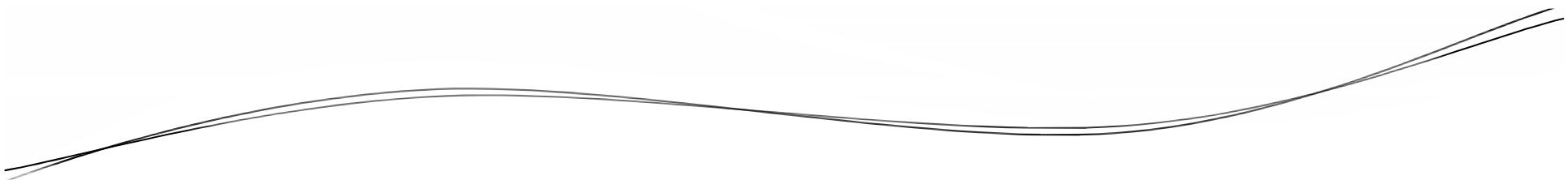


Type 2 Driver

- Native API, partly Java driver.
- It uses a combination of Java as well as Database proprietary standards for implementation.

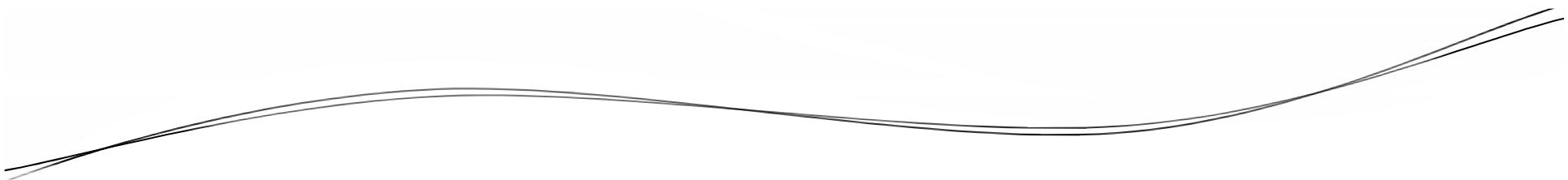
Type 2 Driver





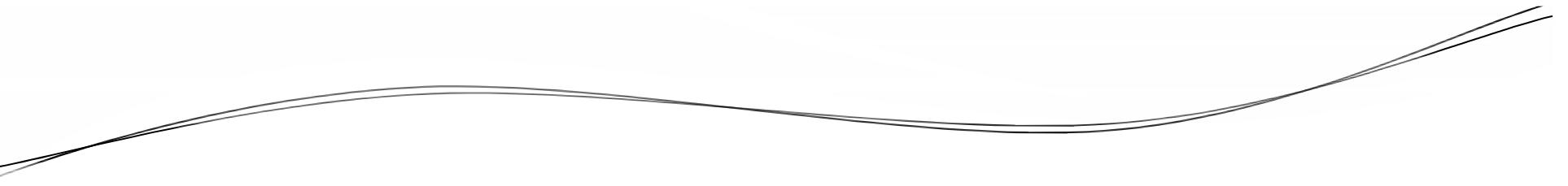
Type 2 Driver

- It does not use any 3rd party library and hence it is platform independent and faster in processing as compared to Type 1.



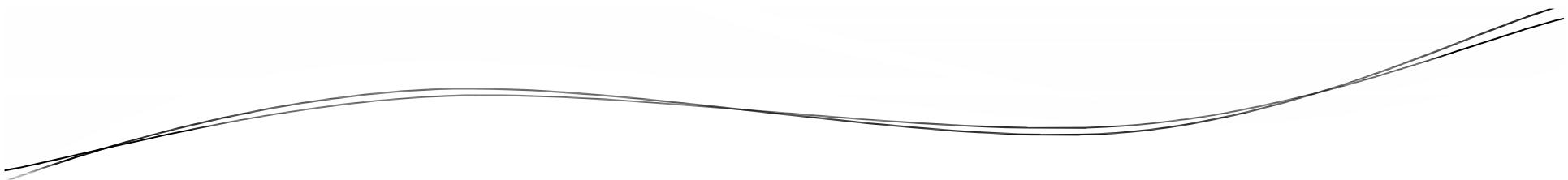
Type 2 Driver

- Since it uses a DB specific native API, the corresponding API must be installed on every client machine.



Type 3 Driver

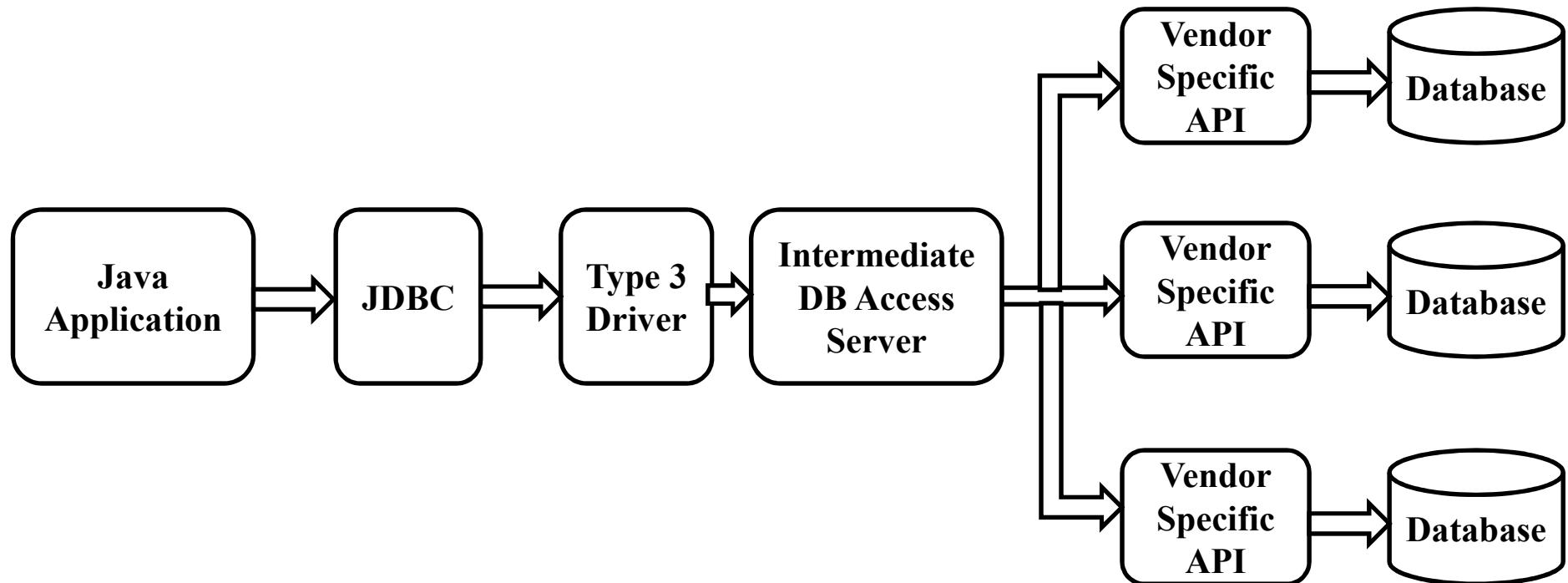
By Rahul Barve

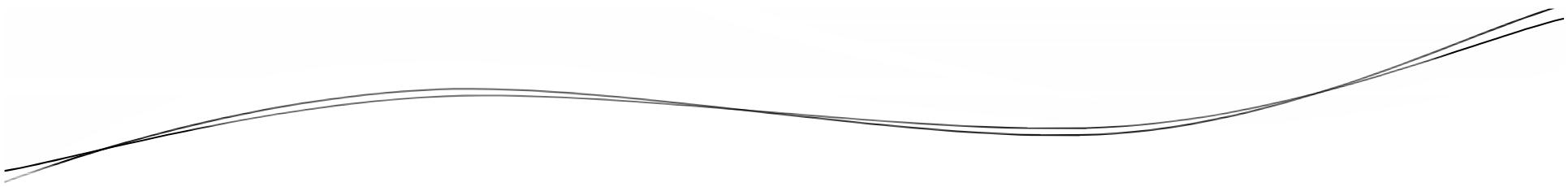


Type 3 Driver

- Net Protocol, Intermediate DB Access Server
- It is used especially when an application needs to connect to multiple databases.

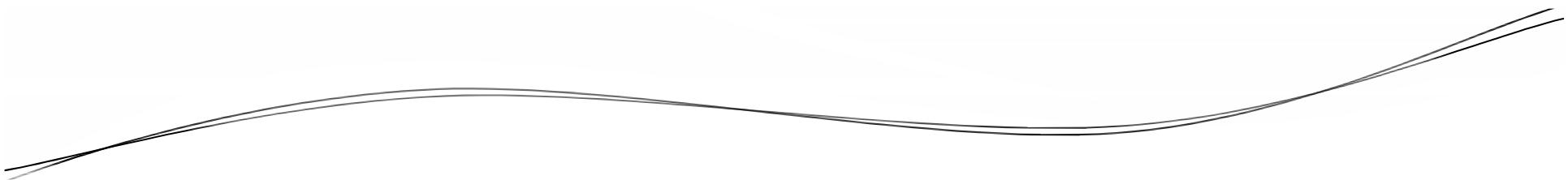
Type 3 Driver





Type 4 Driver

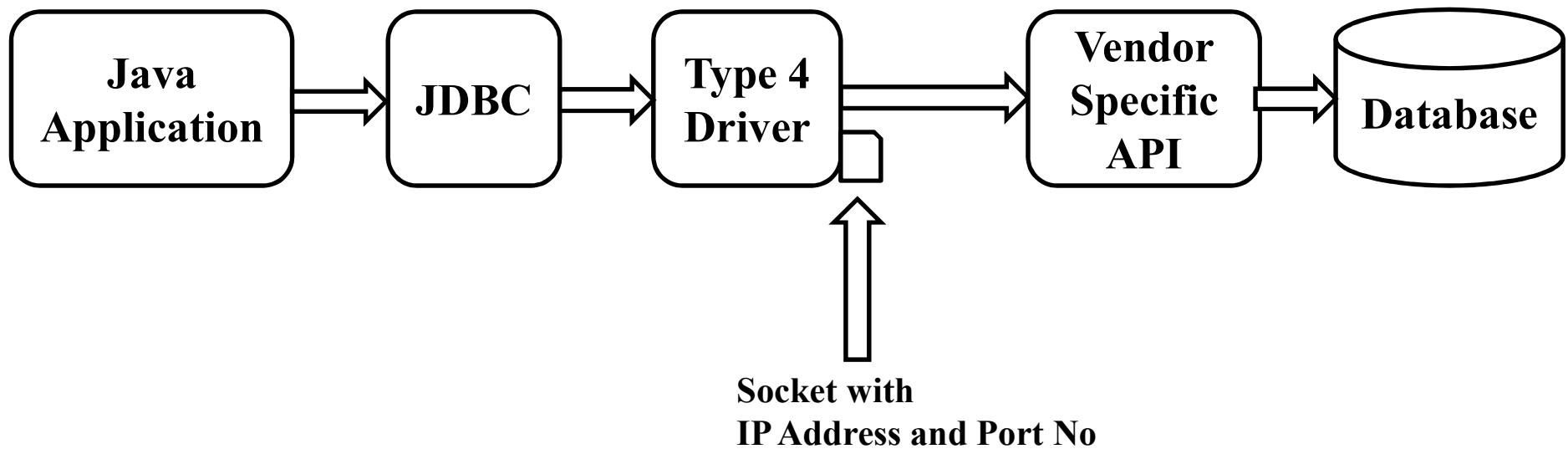
By Rahul Barve

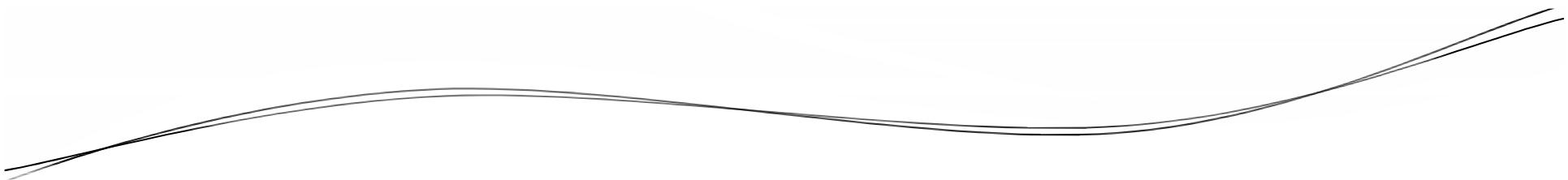


Type 4 Driver

- Database specific, Pure Java Driver.
- Every DB vendor provides its own driver implementation.
- Directly connects to a DB server using TCP/IP socket connections.

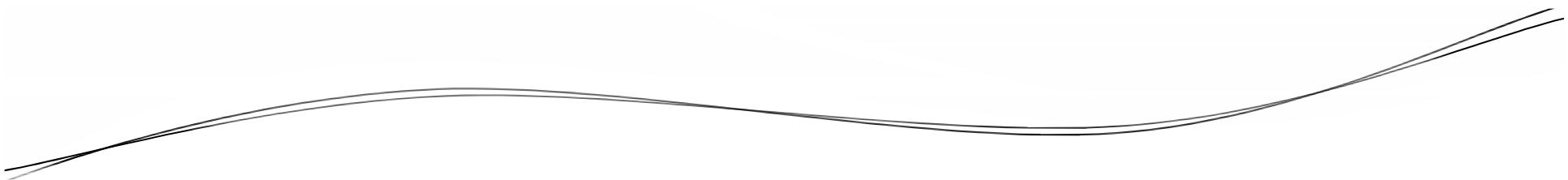
Type 4 Driver





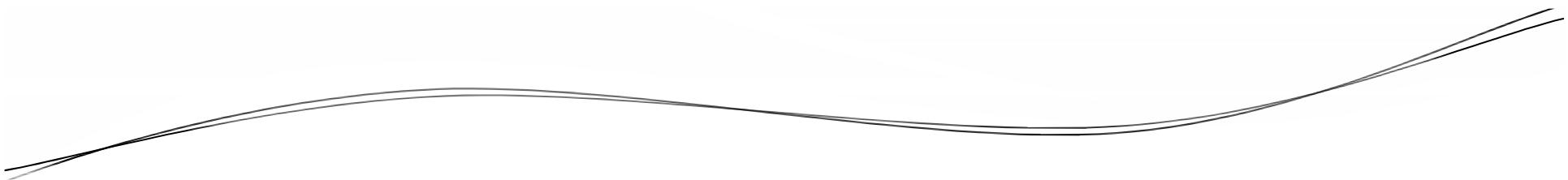
Type 4 Driver

- It is the fastest as compared to Type 1 and Type 2 drivers.
- Platform independent.



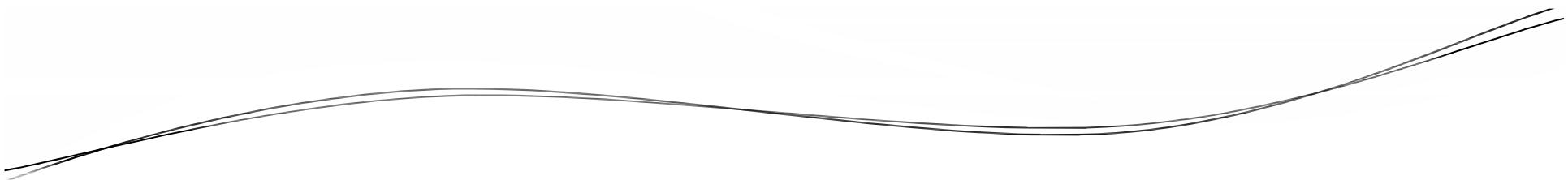
Type 4 Driver

- No configuration is required on the client machine.
- Hence, highly recommended for large scale applications as well as production environment.



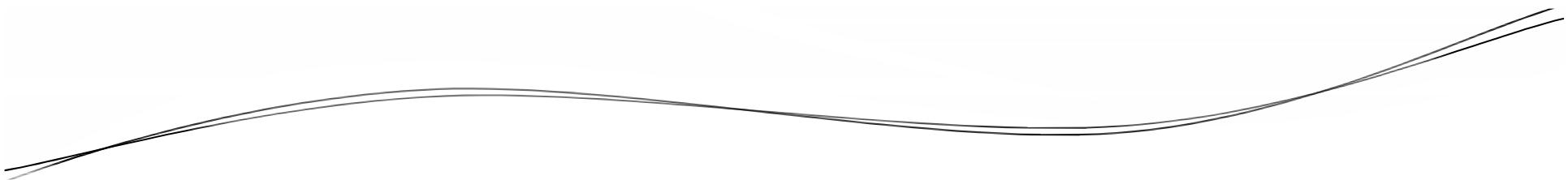
JDBC Core API

By Rahul Barve



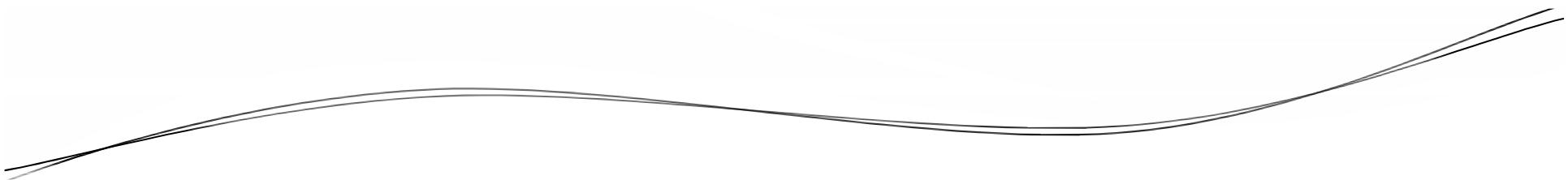
JDBC Core API

- To implement any JDBC program, Java provides an API known as a JDBC API.
- It belongs to a package `java.sql`.



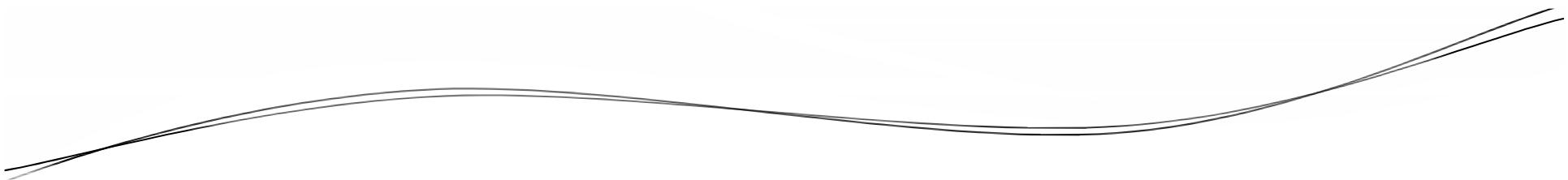
JDBC Core API

- DriverManager
- Driver
- Connection
- Statement
- PreparedStatement
- CallableStatement
- ResultSet



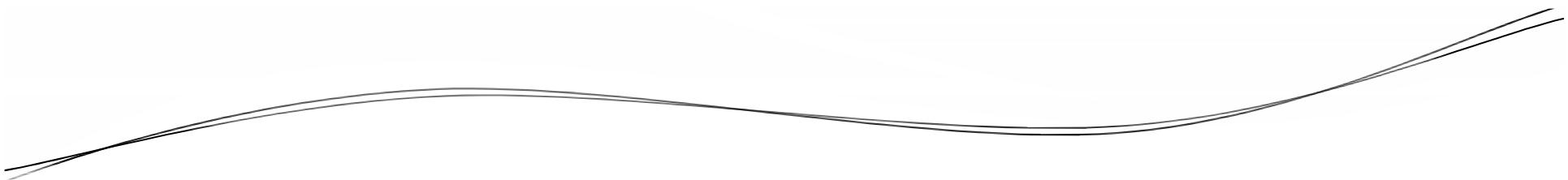
Steps in JDBC Application

By Rahul Barve



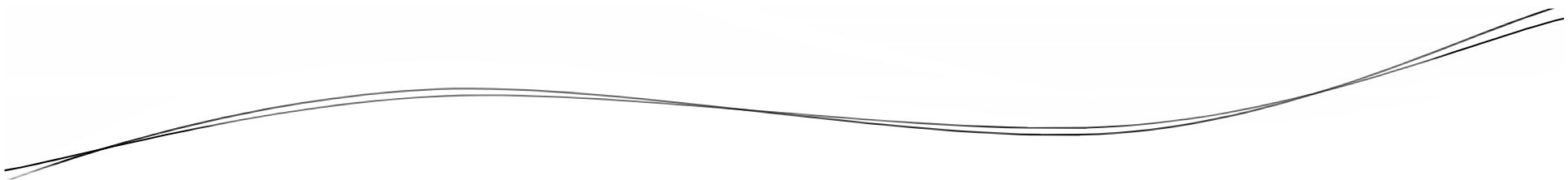
Steps in JDBC Application

- Load the Driver.
- Establish Connection.
- Obtain the Statement.
- Execute SQL query.
- (For SELECT query) Obtain the ResultSet and perform navigation.



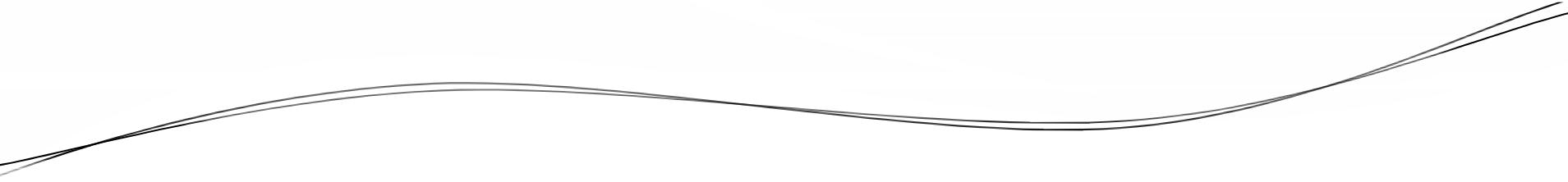
Load the Driver

By Rahul Barve



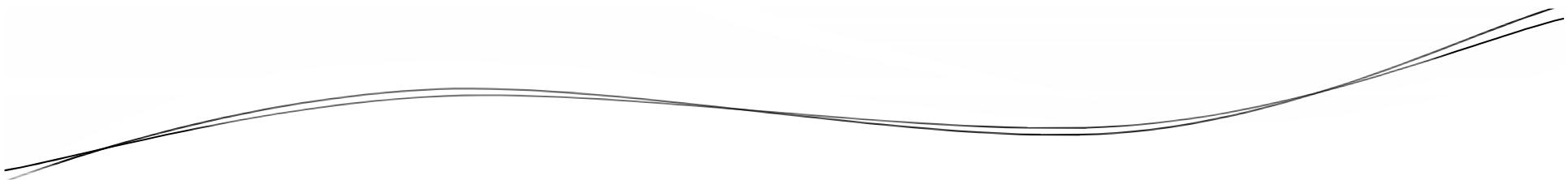
Load the Driver

- A driver can be loaded either by using `Class.forName()` or by creating an object of the driver implementation class.



Establish Connection

By Rahul Barve



Establish Connection

- A Connection to the database can be established either by using a DriverManager class or a Driver interface.

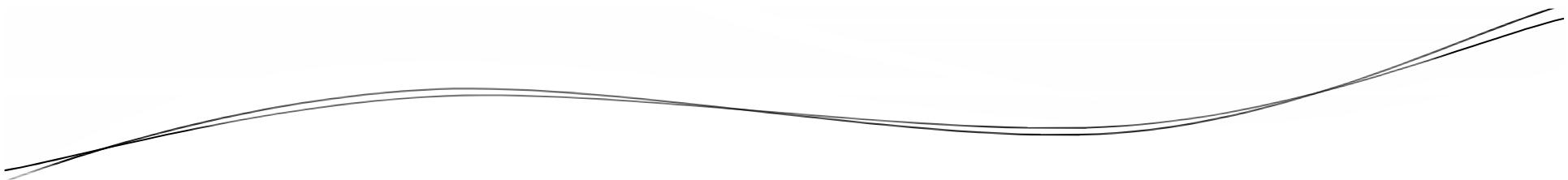
Establish Connection

- E.g.

```
Connection conn =  
    DriverManager.getConnection(...);
```

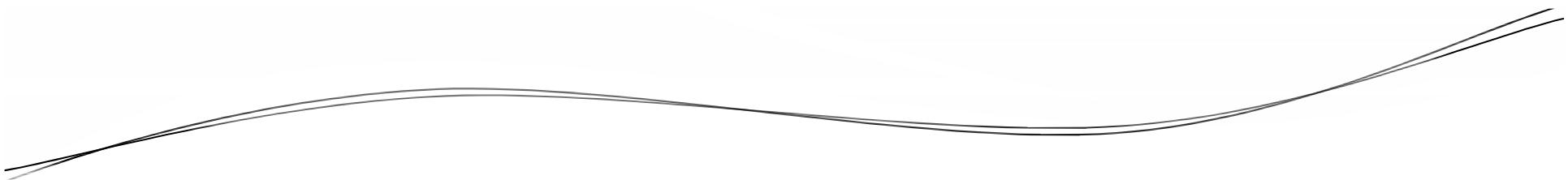
OR

```
Driver dr =  
    new <<DriverImplClassName>>();  
Connection conn = dr.connect(...);
```



Obtain the Statement

By Rahul Barve



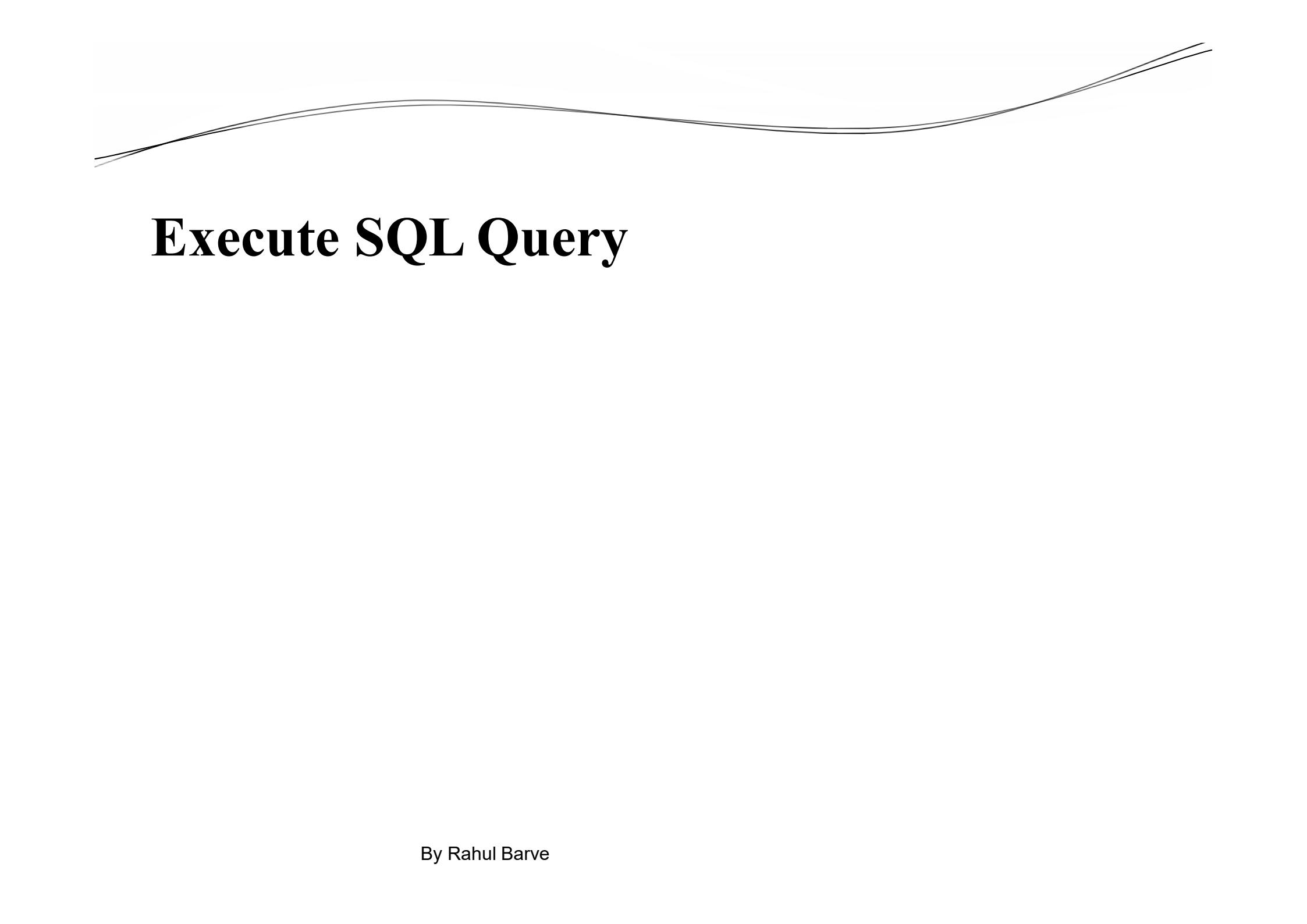
Obtain the Statement

- Once a connection is established, depending upon the type of the operation, a statement needs to be obtained.

Obtain the Statement

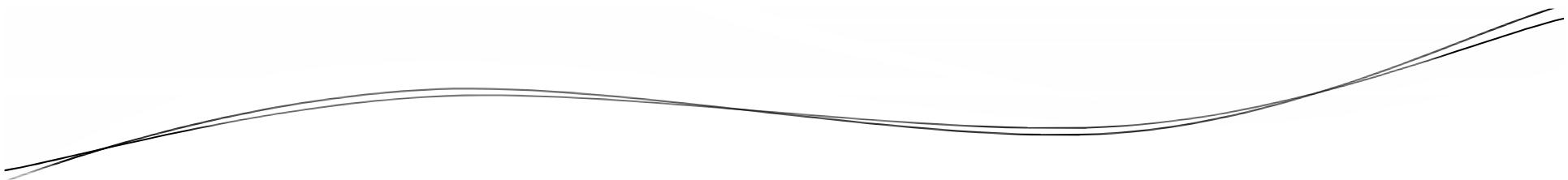
- Statement is used to execute simple queries.

```
Statement stmt =  
    conn.createStatement();
```



Execute SQL Query

By Rahul Barve

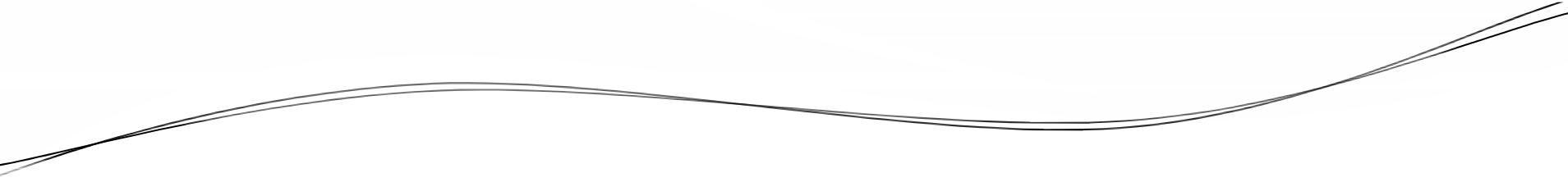


Execute SQL Query

- Statement interface provides relevant methods to execute SQL queries.
- To execute SELECT query, executeQuery() method is used that returns ResultSet.

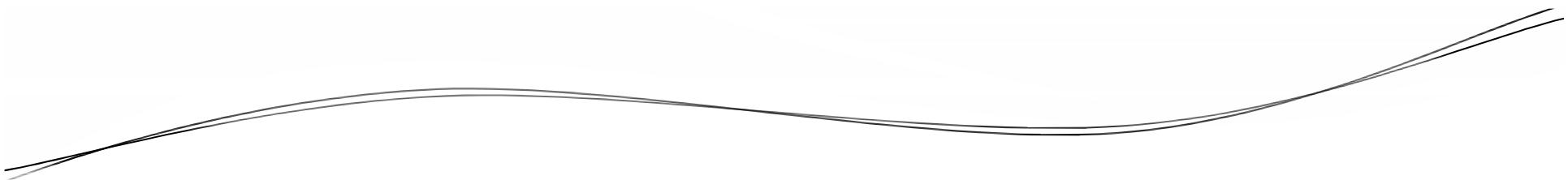
Execute SQL Query

- String sqlQuery = "select";
 ResultSet rs =
 stmt.executeQuery(sqlQuery);



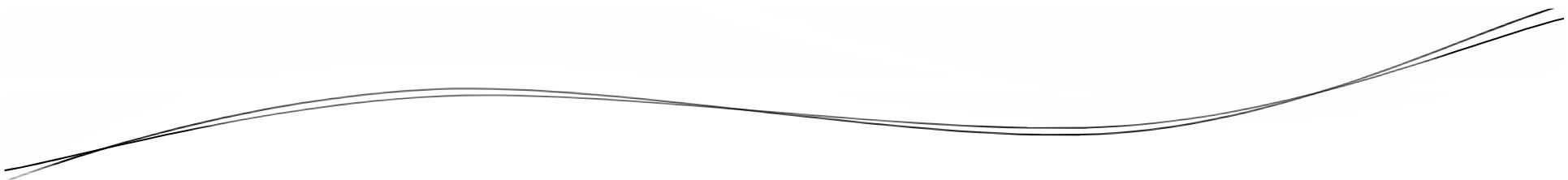
Perform Navigation

By Rahul Barve



Perform Navigation

- ResultSet maintains data fetched from database in a tabular format.
- Every column has a column index and a row has a record position.



Perform Navigation

Before First

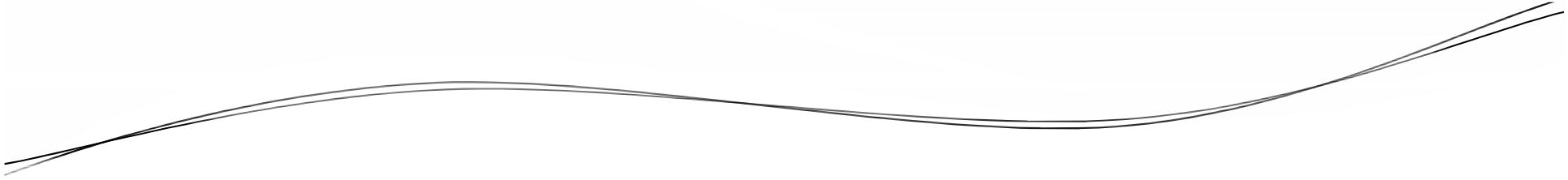
1

2

3

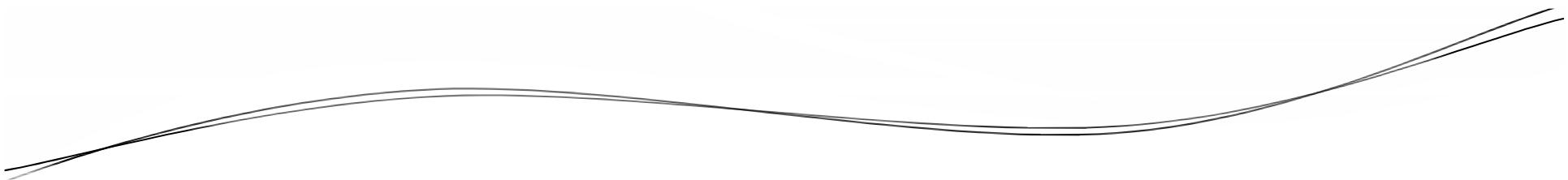
After Last

	1	2	3
1			
2			
3			



Perform Navigation

- By default, the cursor position of ResultSet points to BeforeFirst.
- To move in the forward direction, next () method is used.

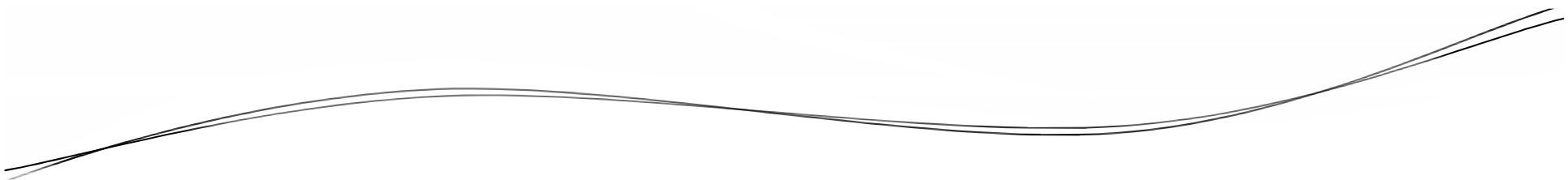


Parameterized Queries

By Rahul Barve

Parameterized Queries

- A query may accept parameters.
- To execute parameterized queries, PreparedStatement interface is used.



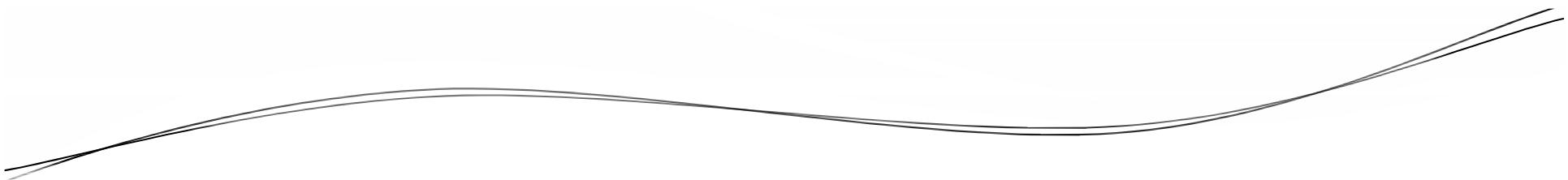
Parameterized Queries

- Queries created using PreparedStatement are compiled once, hence are called as precompiled queries.

Parameterized Queries

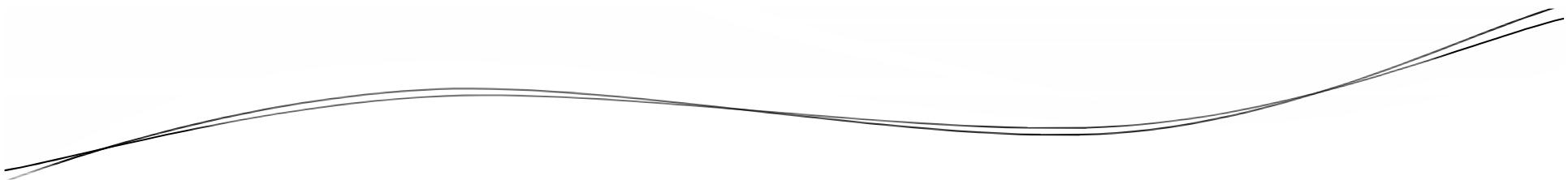
- E.g.

```
String sqlQuery =  
"select ... where deptno in (?, ?)";  
PreparedStatement pstmt =  
conn.prepareStatement(sqlQuery);
```



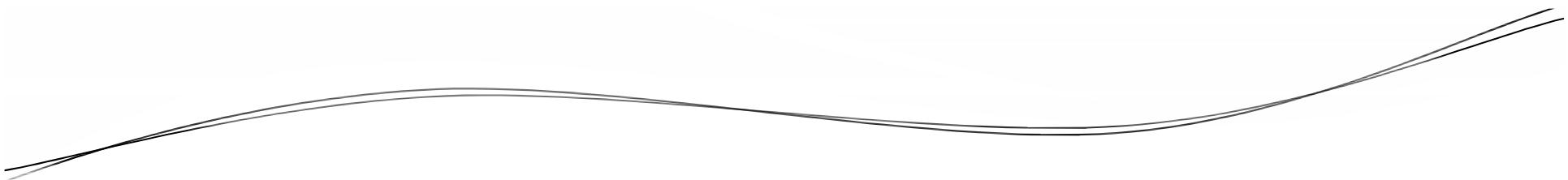
DML Queries

By Rahul Barve



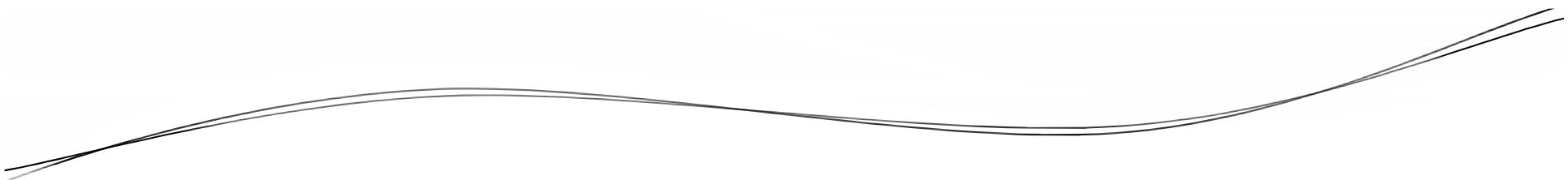
DML Queries

- To execute SELECT queries, executeQuery() method is used whereas to execute DML queries, executeUpdate() method is used.



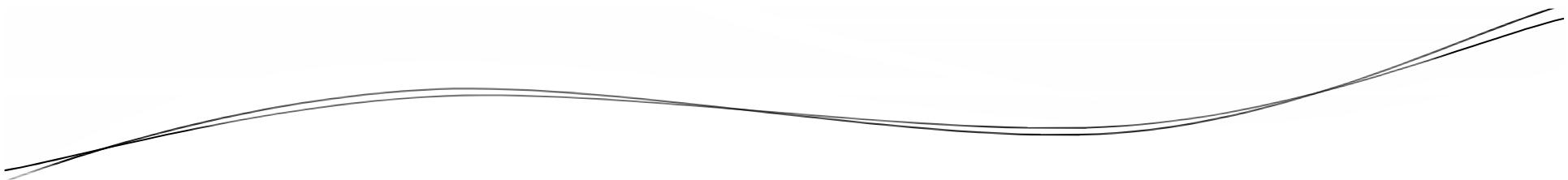
Transaction Management

By Rahul Barve



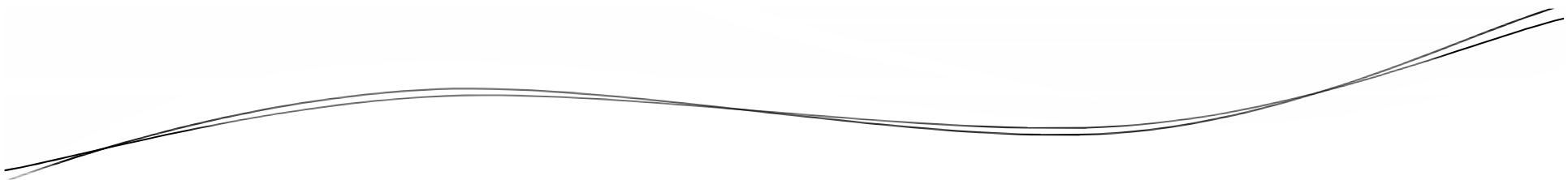
Transaction Management

- Transaction Management is a very important activity in an application development.
- Transaction is a set of operations that must execute in a single unit.



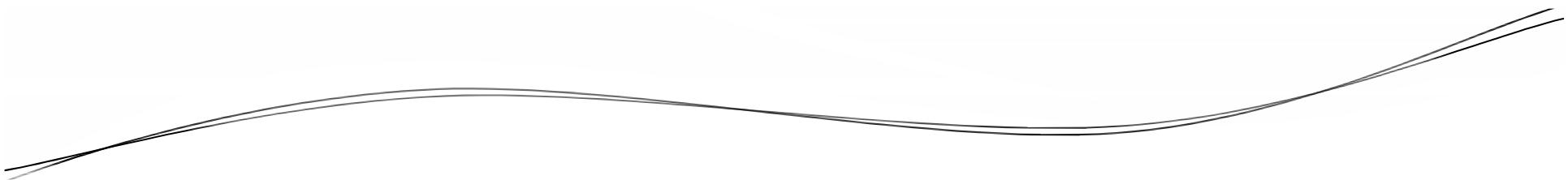
Transaction Management

- Updates made from Java application to database are committed by default.
- Once, changes are committed, cannot be rolled back.



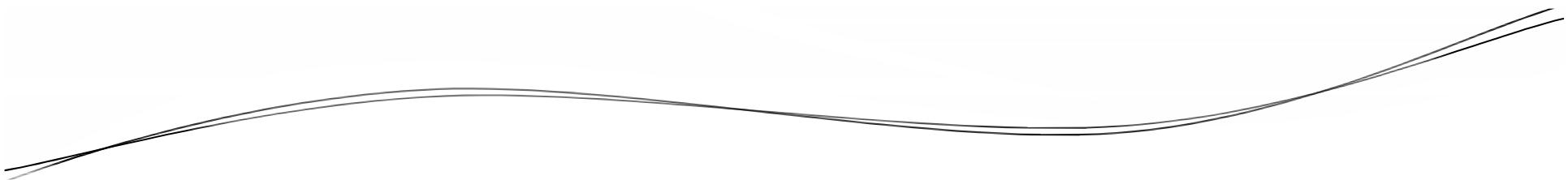
Transaction Management

- To manage the transactions, auto-commit must be disabled.
- This is done by using `setAutoCommit(false)` on the `Connection` object.



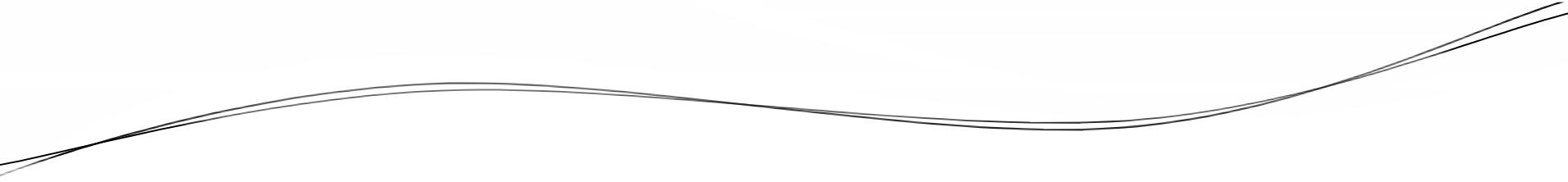
Transaction Management

- Once auto-commit is disabled, it is possible to commit or rollback the transactions by using `commit()` or `rollback()` methods respectively.

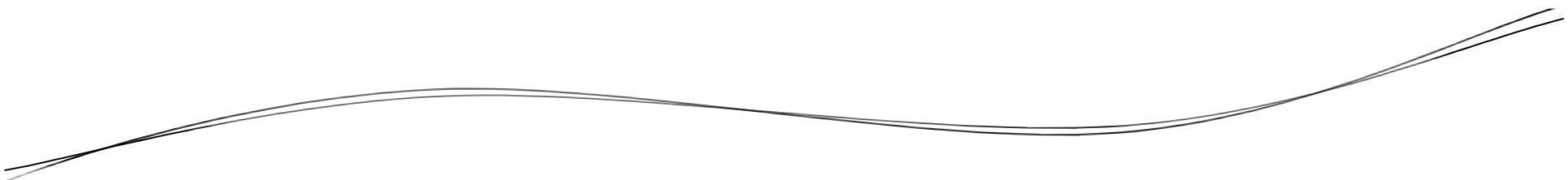


Lets Summarize

- What is JDBC
- Why JDBC
- JDBC Drivers
- JDBC Core API
- Executing Simple Queries
- Executing Parameterized Queries
- Transaction Management

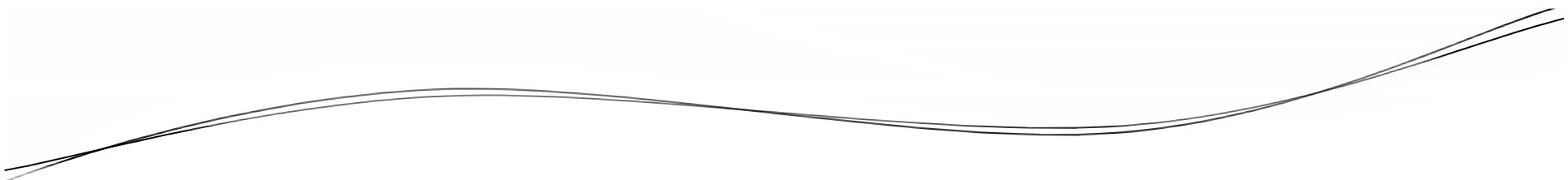


Servlet

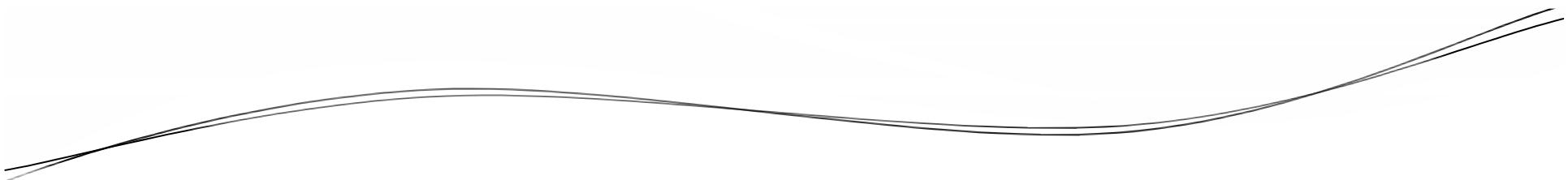


Objectives

- HTTP Basics
- Introduction to Servlets
- Implementing Servlets
- Life Cycle
- Request Handling

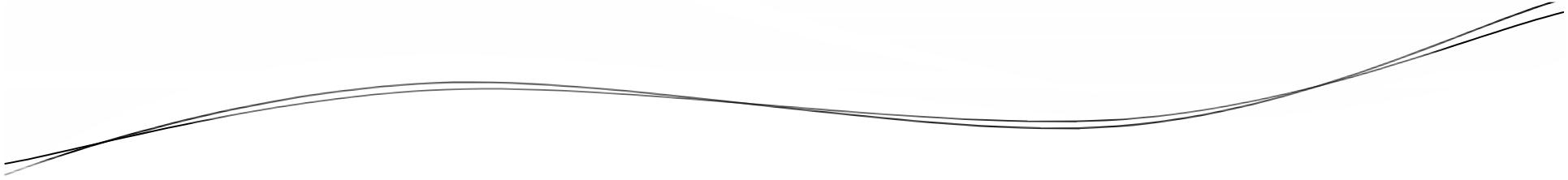


HTTP



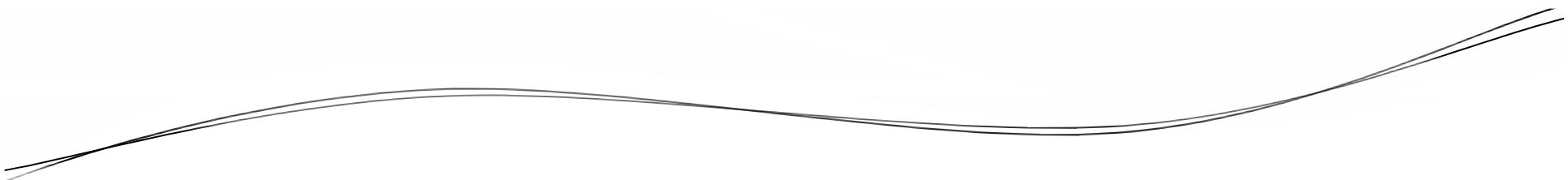
HTTP

- HTTP stands for Hyper Text Transfer Protocol.
- HTTP is a stateless protocol or request-response protocol.

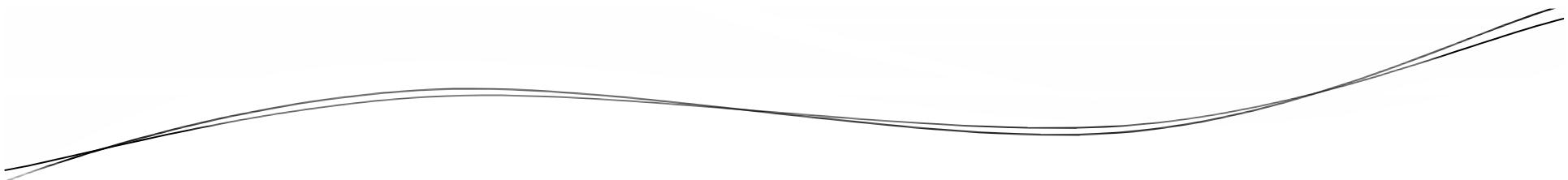


HTTP

- Does not maintain any conversational state between the 2 requests.
- Cannot recognize the client.
- The most commonly used protocol in Web Application.

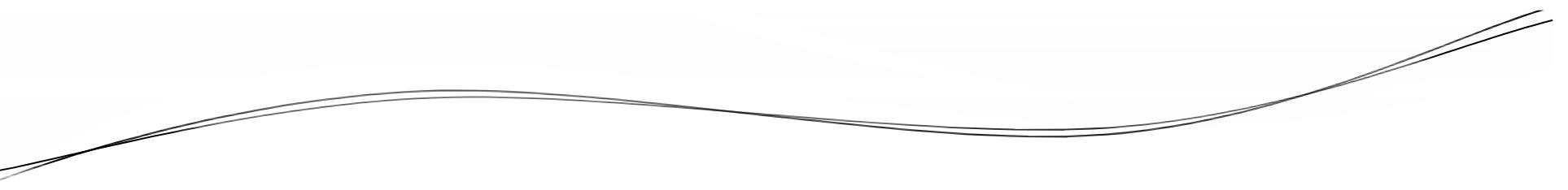


What is Servlet

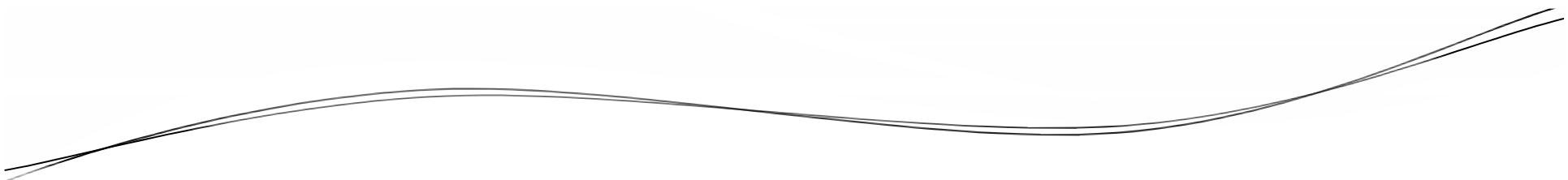


What is Servlet

- Servlet is a component that is used to extend the functionality of web server.
- A component that resides on server side and performs server side processing.
- Used to generate dynamic web contents.

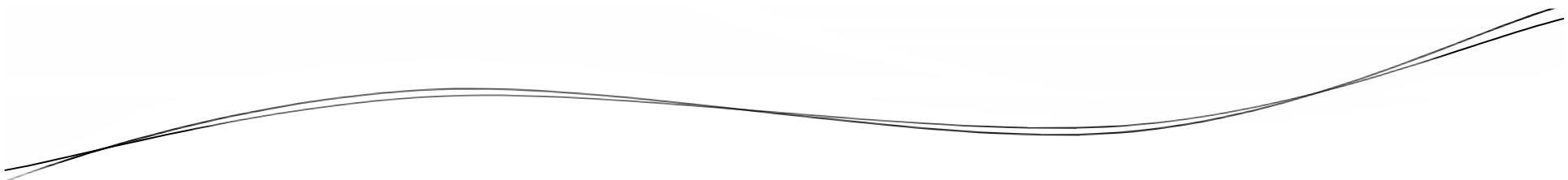


Why Servlet



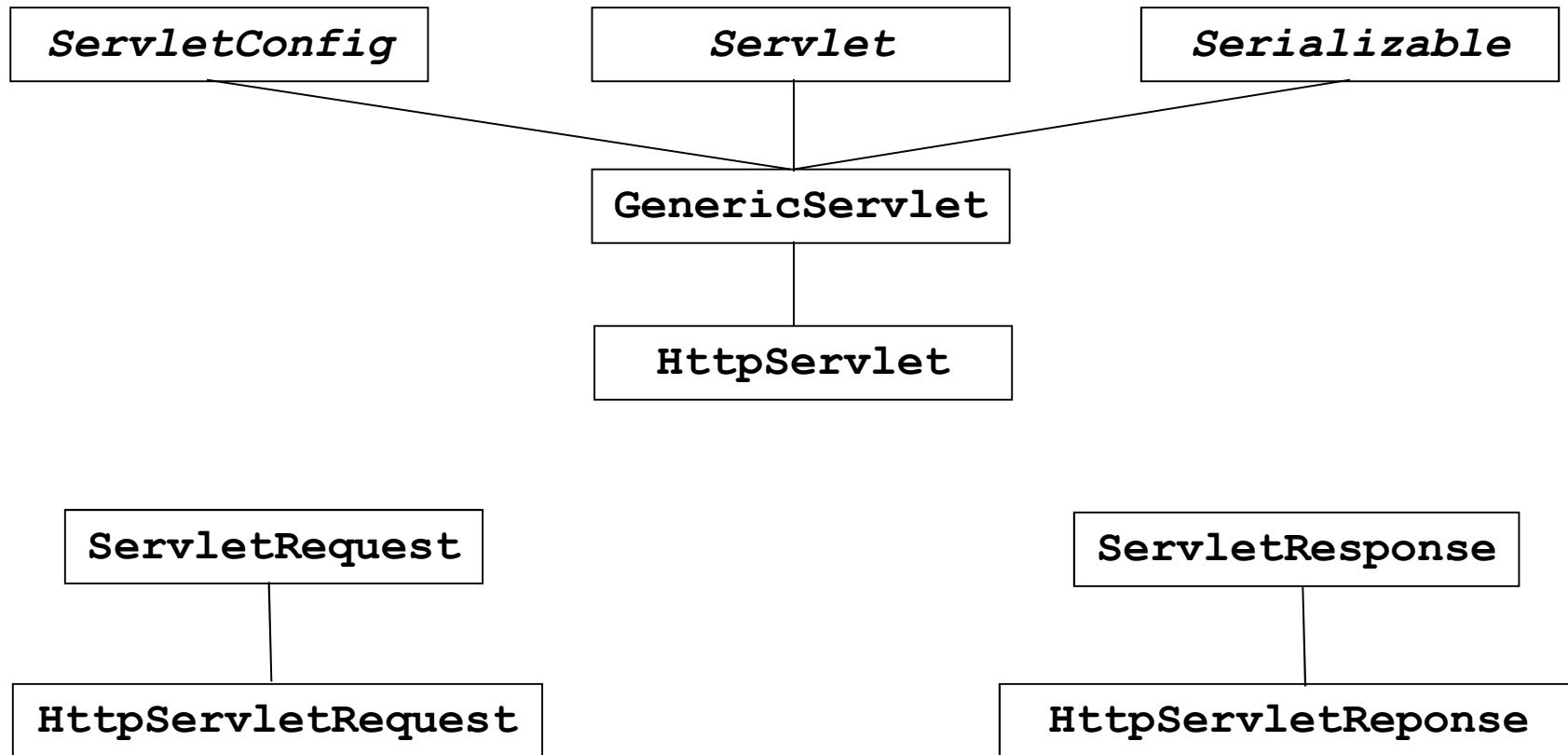
Why Servlet

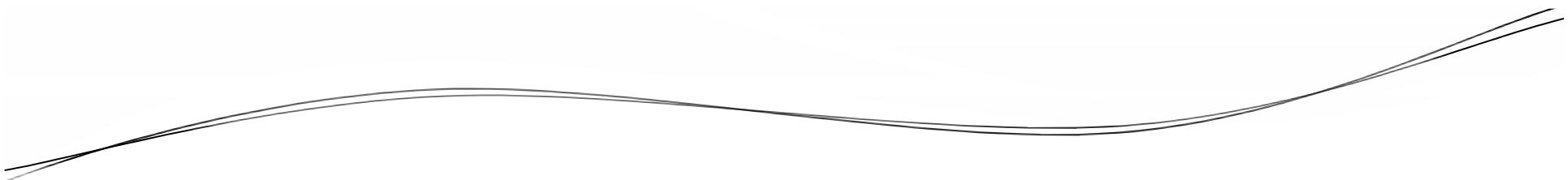
- Servlets are written in Java language, thus inherit all the features of Java.
- Portable
- Secured
- Platform Independent



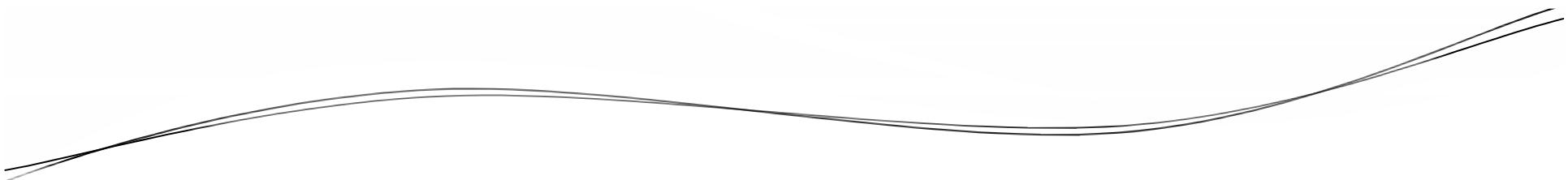
Servlet API

Servlet API



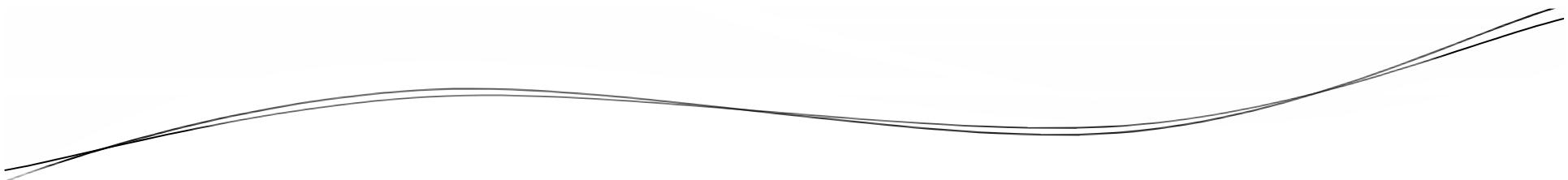


Servlet Life Cycle



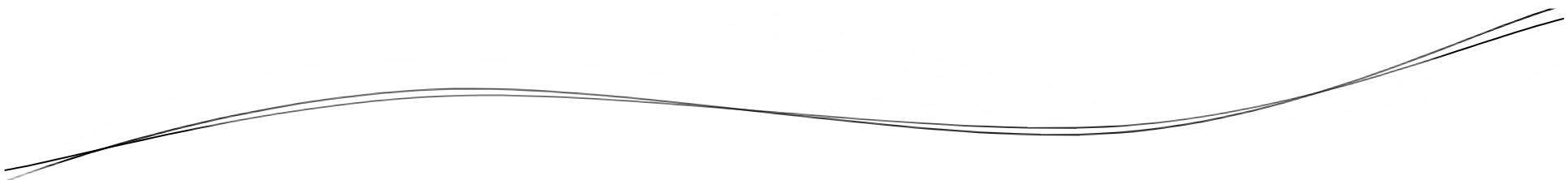
Servlet Life Cycle

- Life Cycle of Servlet consists of 3 stages:
 - Instantiation and Initialization
 - Service
 - Destroy



Servlet Life Cycle

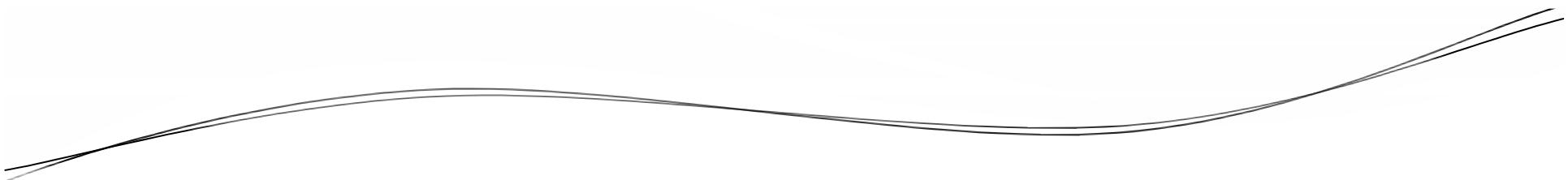
- There are 3 life cycle methods:
 - `init()`
 - `service()`
 - `destroy()`



ServletConfig

ServletConfig

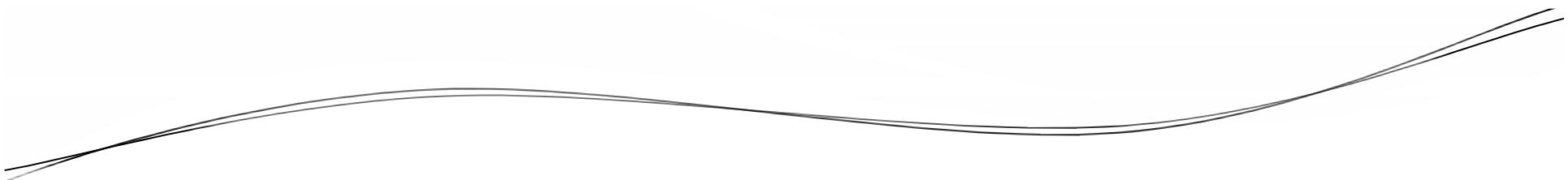
- An object of ServletConfig is associated with a servlet.
- Stores configuration specific information related to the servlet.
- Can be used to retrieve initial parameters.



ServletContext

ServletContext

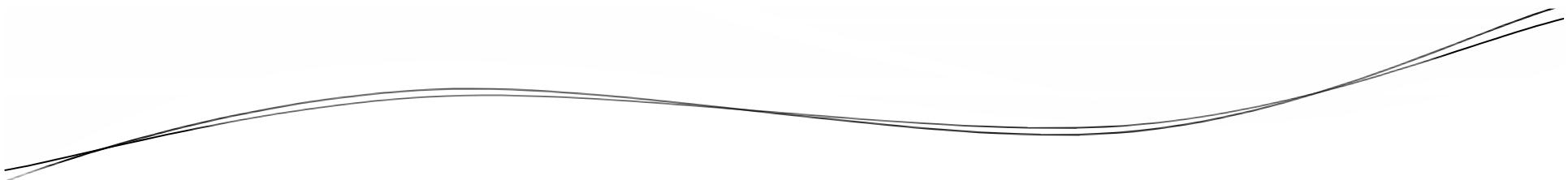
- An object of ServletContext is created per application.
- Thus, useful to handle the application level information.



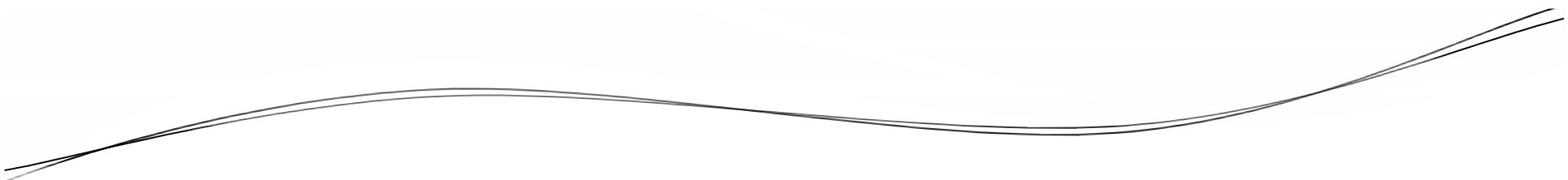
ServletContext

- **Useful Methods:**

- `public void setAttribute(String, Object);`
- `public Object getAttribute(String);`



HTML Form Processing



HTML Form Processing

- In a web application, end user enters data using some HTML form.
- Once, SUBMIT is clicked, request is made to the server and it is to be processed by some server side component.

HTML Form Processing

- E.g. User validation using Login page, User registration using registration page.
- This is done using action attribute of the HTML <form> element.

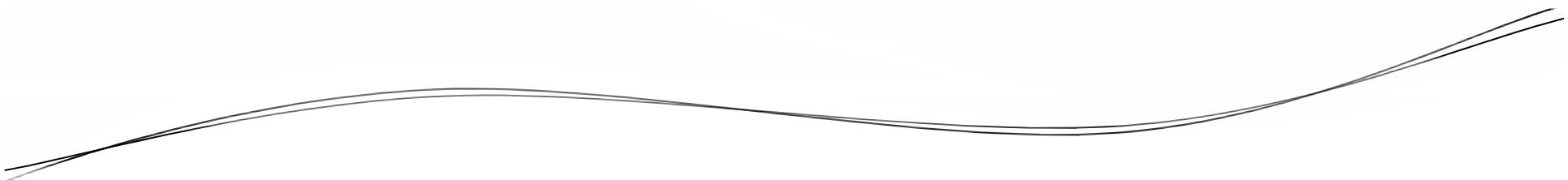
Difference between GET and POST

GET

- Request parameters are appended to URL.
- Limitation on data transfer. Generally 8kb.
- Limitation on length of the URL: 255 characters

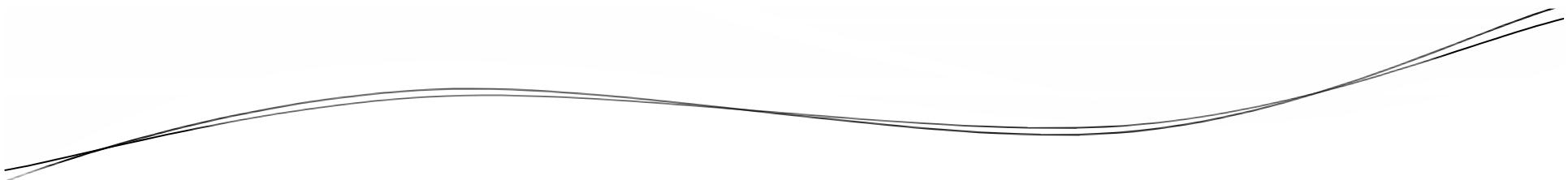
POST

- Request parameters are sent with the page body.
- There is no limitation on data transfer.
- There is no limitation on URL length.

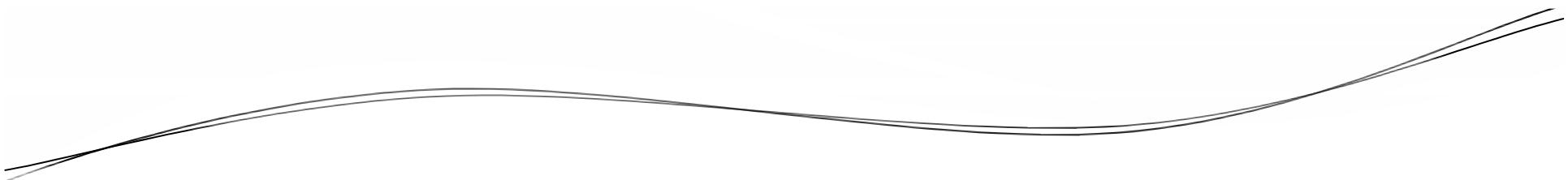


Let's Summarize

- HTTP Basics
- What is Servlet and its Need
- Servlet API
- Implementing Servlets
- Servlet Life Cycle
- HTML Form Processing

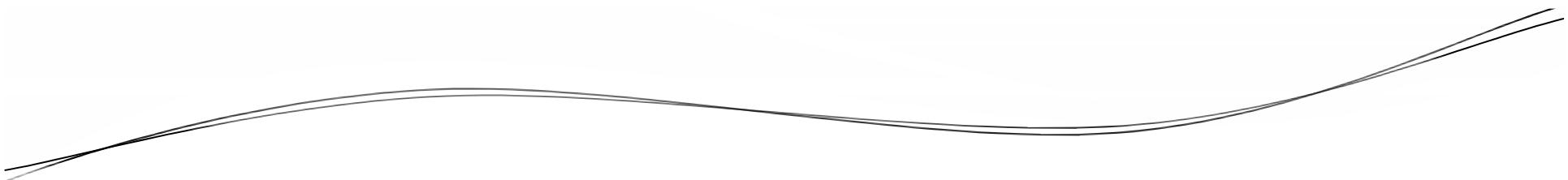


Servlet Part 2

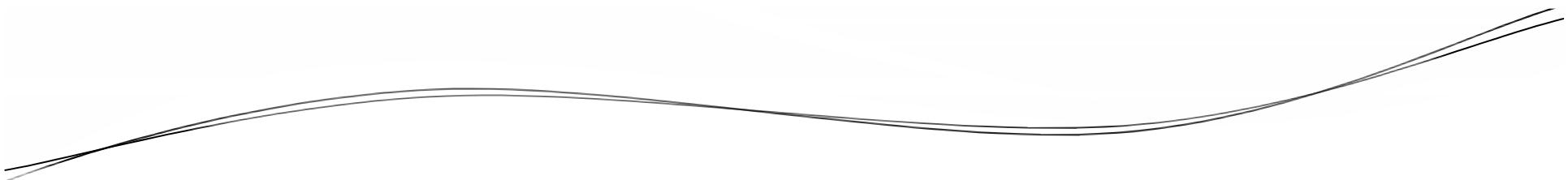


Objectives

- Collaboration
- Session Management

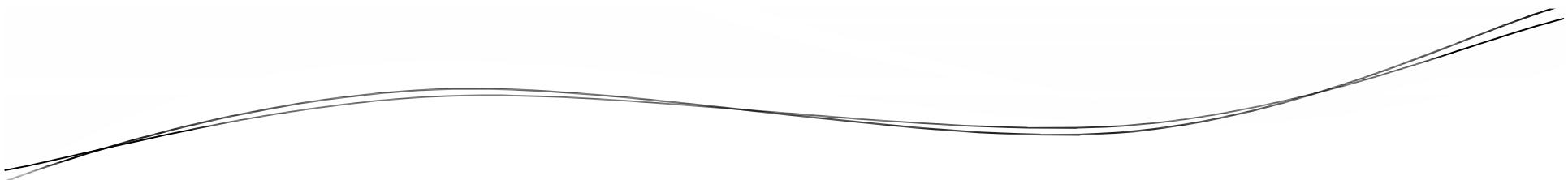


Collaboration



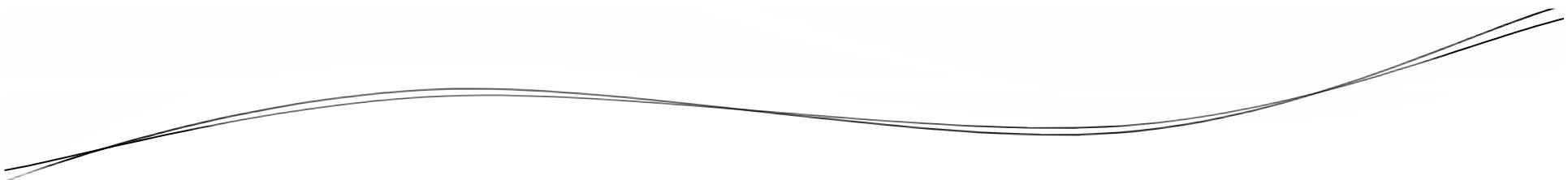
Collaboration

- When 2 components of same web application are interacting with each other, that process is known as collaboration.

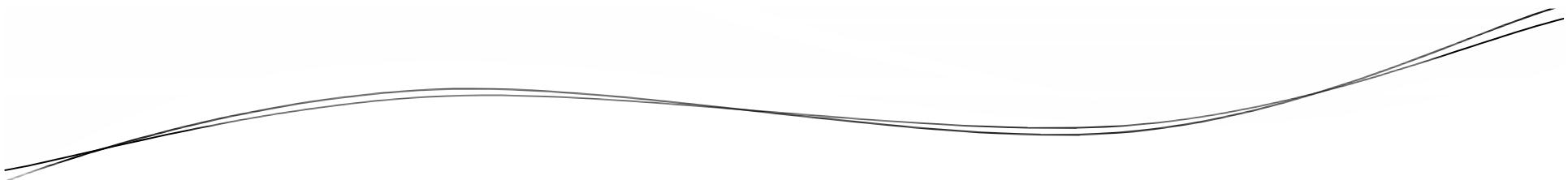


Collaboration

- Benefits
 - Modularity
 - Reusability



RequestDispatcher



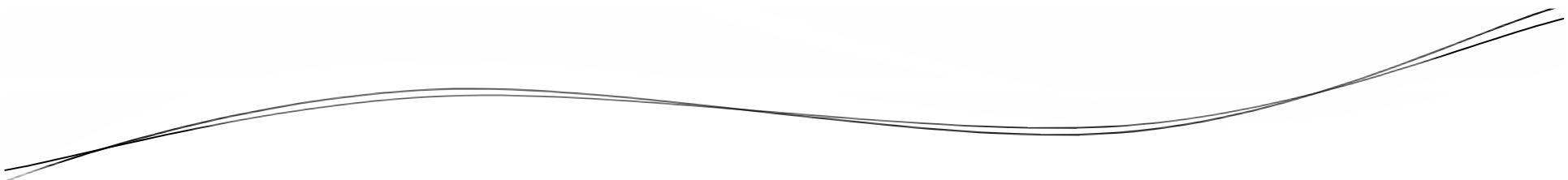
RequestDispatcher

- Used to achieve collaboration between the components running within the same web application.

RequestDispatcher

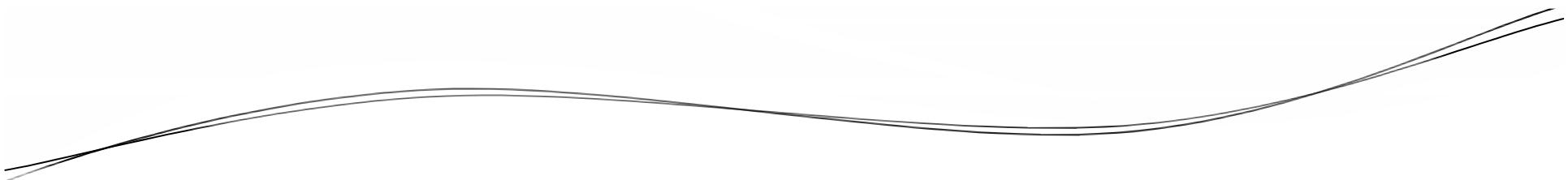
- Methods:

- public void forward(ServletRequest, ServletResponse);
- public void include(ServletRequest, ServletResponse);

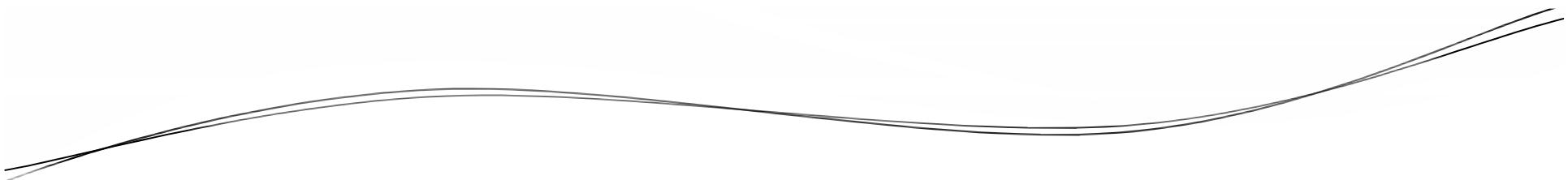


RequestDispatcher

- forward()
 - A called resource generates the final response back to the client.
- include()
 - A calling resource generates the final response back to the client.

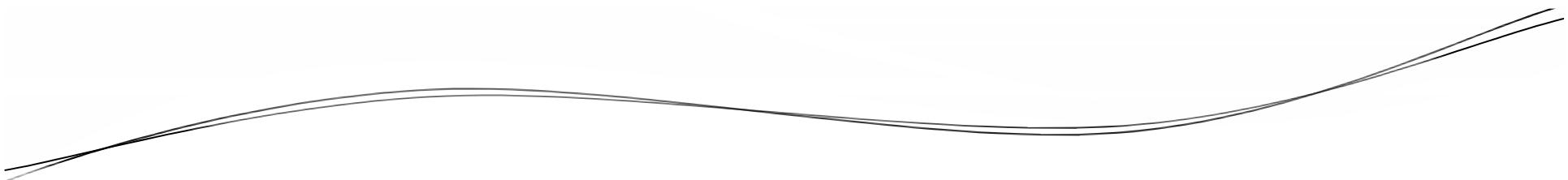


Session Management



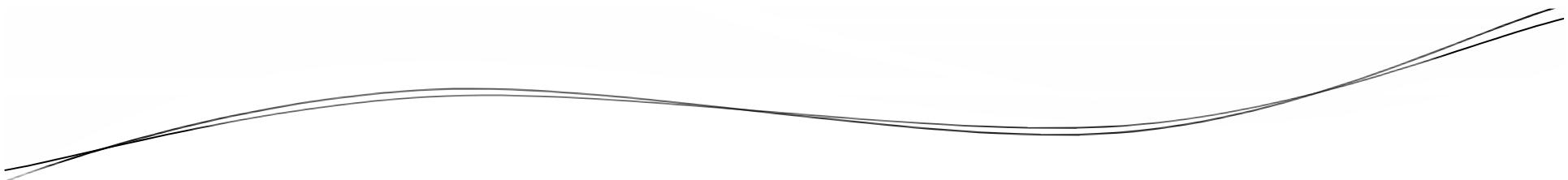
Session Management

- HTTP is a stateless protocol.
- In a web application, an end user can make some transaction through one or multiple requests.



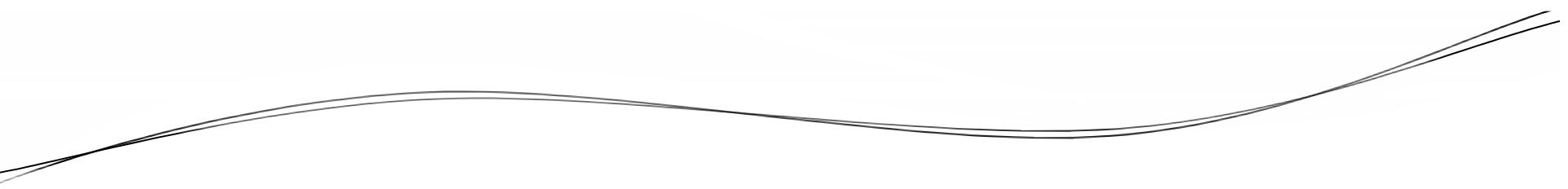
Session Management

- During this, server needs to maintain a conversational state along with the client.
- This technique is known as session tracking.

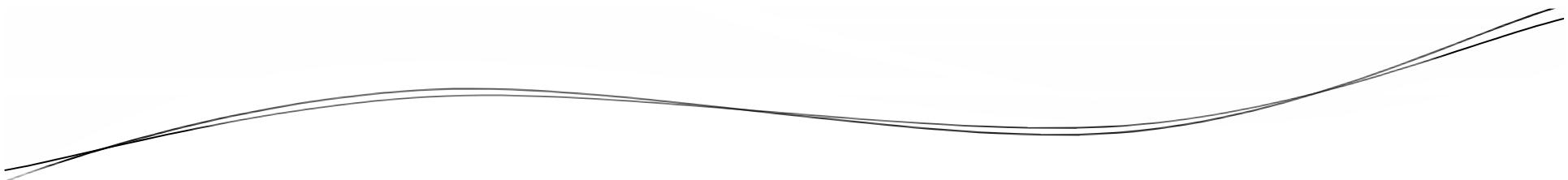


Session Management

- Different methods used for Session Tracking:
 - URL Rewriting
 - Hidden Fields
 - Cookies
 - Servlet API - HttpSession

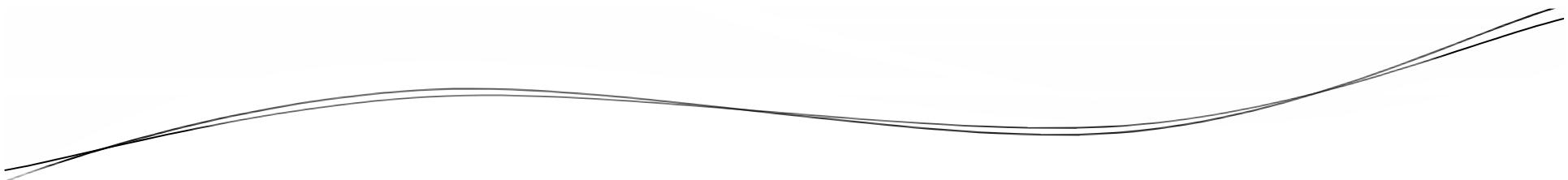


HttpSession



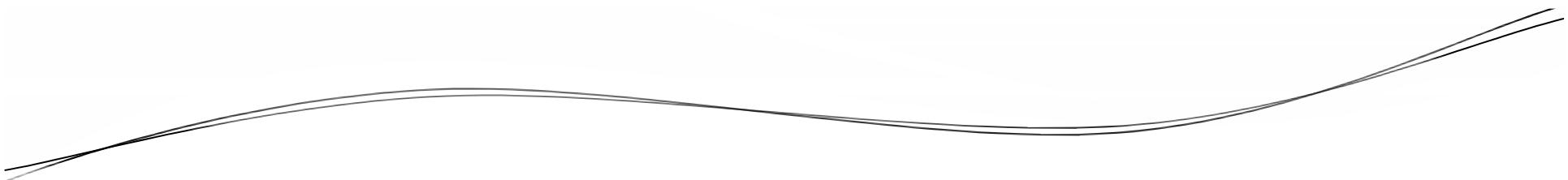
HttpSession

- A Servlet API that is used to handle Session Tracking.
- HttpServletRequest is used to obtain the object of HttpSession.
 - getSession()
 - getSession(boolean)



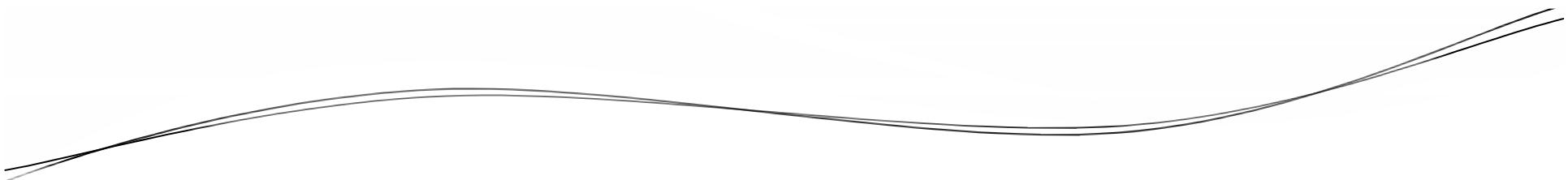
HttpSession

- Important Methods:
 - setAttribute()
 - getAttribute()
 - isNew()
 - setMaxInactiveInterval()
 - invalidate()

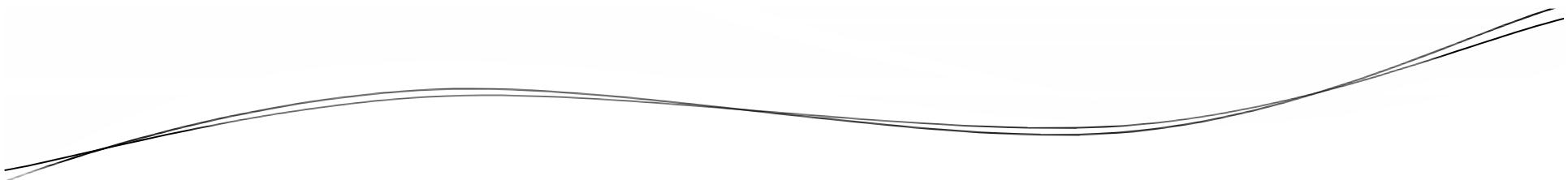


Let's Summarize

- Collaboration
- Session Management

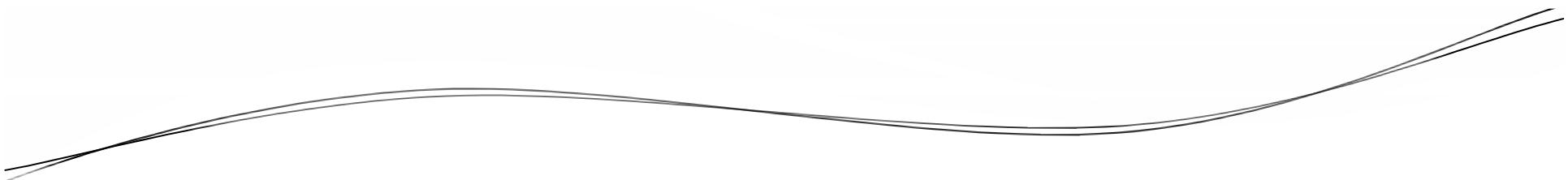


JSP

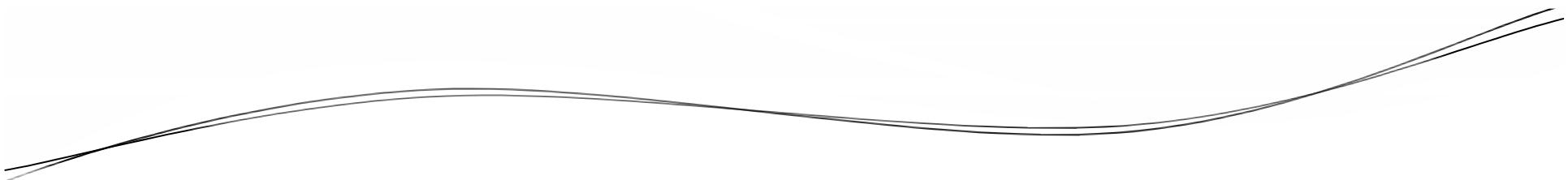


Objectives

- Understand: What is JSP, Why JSP
- Explain: Life Cycle of JSP
- Working with JSP Tags
- Working with Implicit Objects

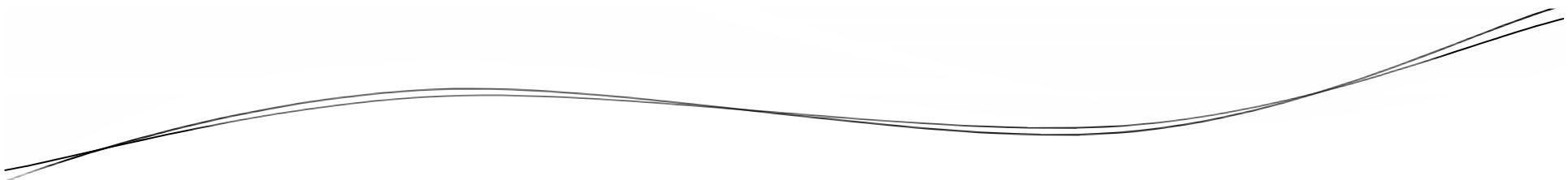


What is JSP



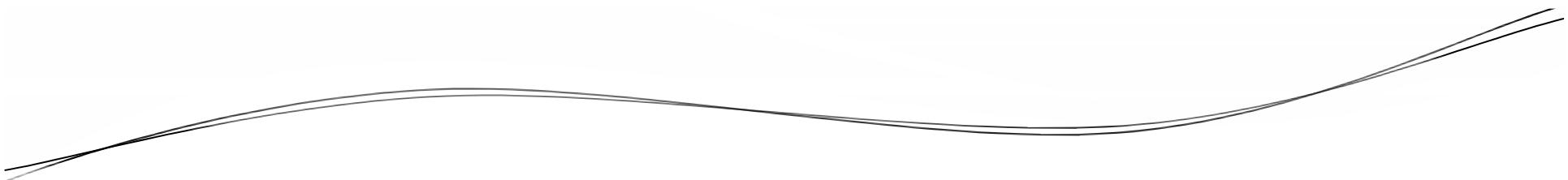
What is JSP

- JSP stands for Java Server Pages.
- JSP is a server side component that is used to extend the functionality of web server.
- Used to generate dynamic web content.



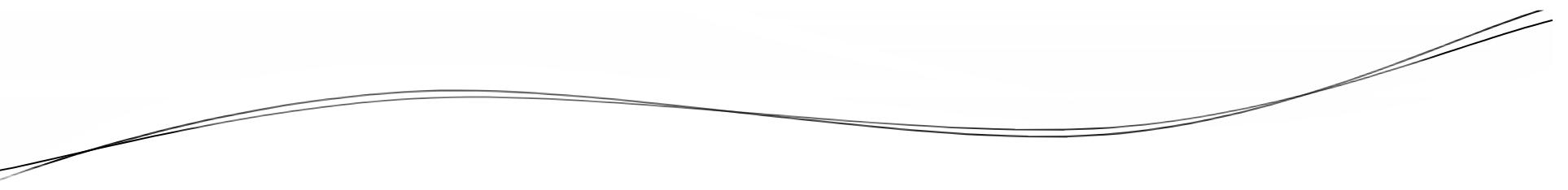
Why JSP

- Allows developers to concentrate on Presentation rather than Processing.
- Designers without knowing Java, still can develop elegant web pages.

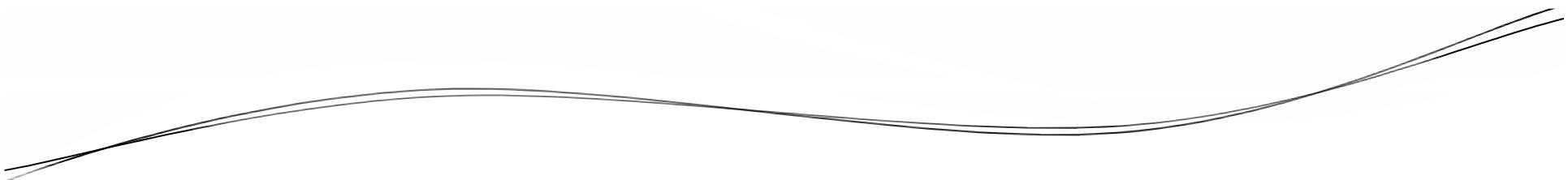


JSP Life Cycle

- There are 3 life cycle methods:
 - `jspInit()`
 - `_jspService()`
 - `jspDestroy()`

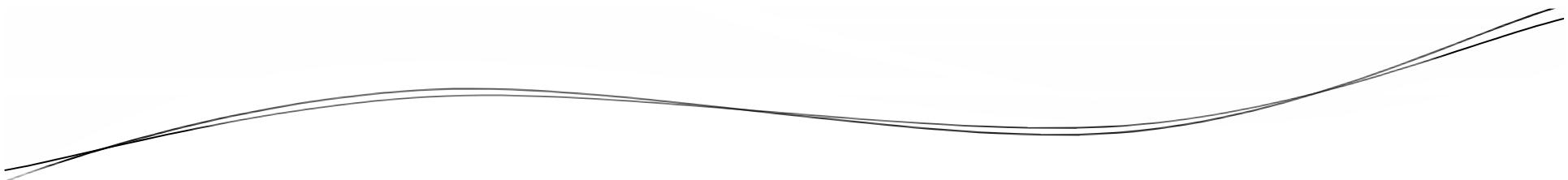


JSP Tags



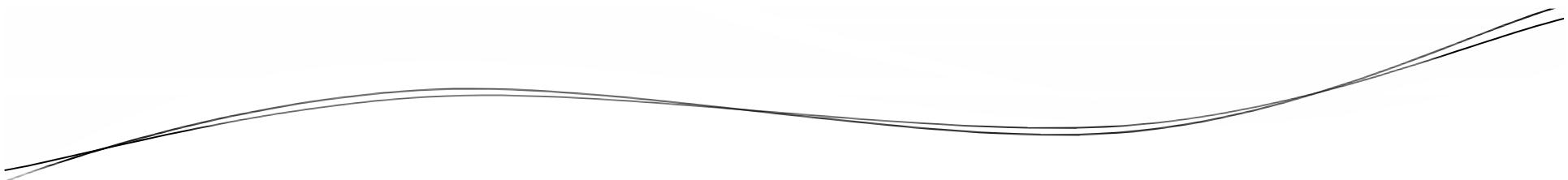
JSP Tags

- JSP Specification supports 3 types of tags:
 - Directives
 - Scripting Elements
 - Standard Actions



Directives

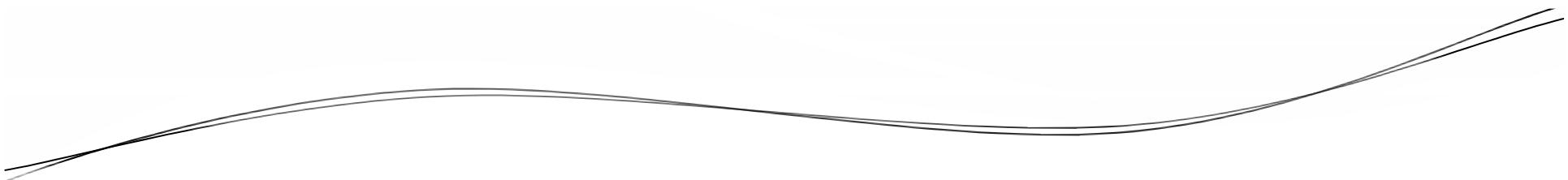
- Directives are divided into 3 categories:
 - page
 - include
 - taglib



Page Directive

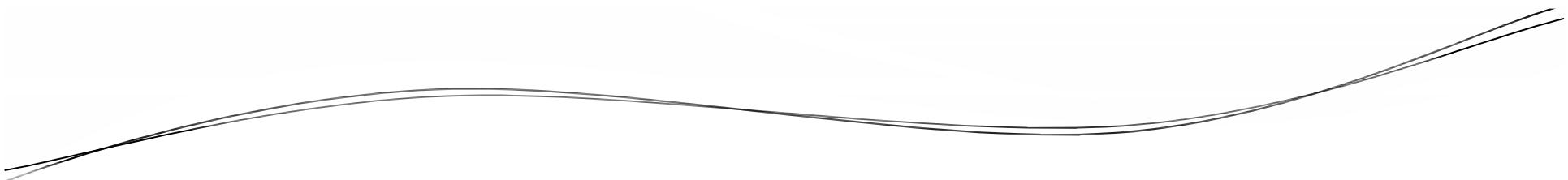
- Used to specify some information about the page.
- Syntax:

```
<%@page attr="value" ...%>
```



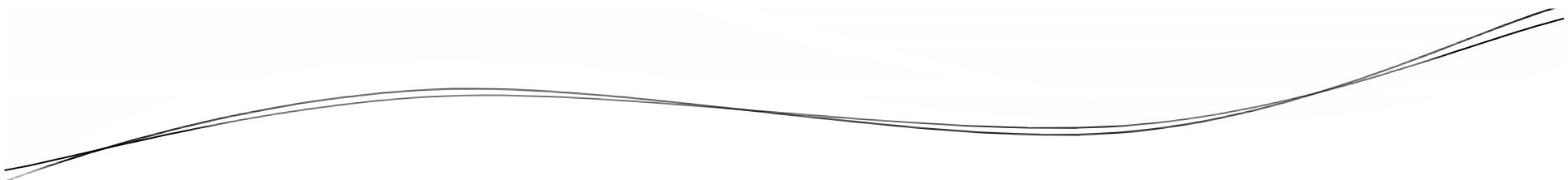
Page Directive

- Attributes:
 - language
 - extends
 - import
 - session
 - isThreadSafe
 - errorPage
 - isErrorPage



Page Directive

- language
 - Specifies scripting language of the JSP page.
 - Possible Values: java
 - Default Value: java



Page Directive

- extends
 - Specifies fully qualified name of the class from which JSP specific Servlet implementation class will inherit from.

Page Directive

- import
 - Specifies fully qualified names of the classes or interfaces to be imported from the package other than `java.lang`.
 - E.g.

```
<%@ page import =
"java.util.Vector, java.io.File" %>
```

Page Directive

- session
 - Specifies whether the page participates in a session or not and therefore whether the implicit object session will be available or not.
 - Possible Values: true / false
 - Default Value : true

Page Directive

- `isThreadSafe`
 - Tells container whether to make the JSP page thread safe or not.
 - It is in contrast to the thread safety concept of multithreading.
 - Possible Values: `true` / `false`
 - Default Value: `true`

Page Directive

- `errorPage`
 - Specifies the name of the JSP page to which control is to be diverted if the current JSP page contains some Java code that fires an exception.

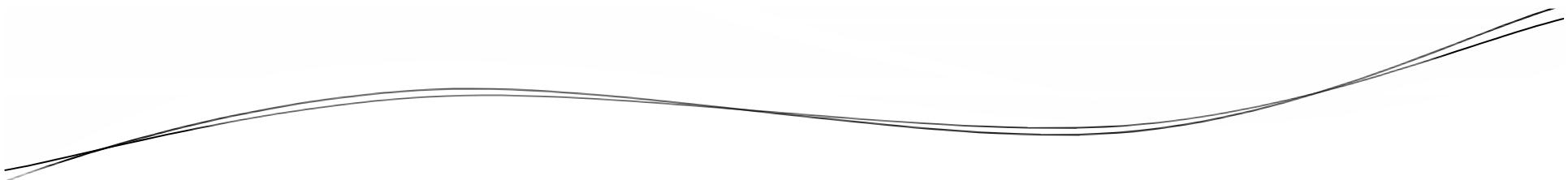
Page Directive

- isErrorPage
 - Indicates whether the JSP page is an error page or not. If so, an implicit object exception becomes available in the page.
 - Possible Values: true / false
 - Default Value: false

Include Directive

- Used to include resources like HTML, JSP or Text files in JSP.
- Syntax:

```
<%@include file = "<filename>"%>
```



Scripting Elements

- Scripting Elements are divided into 3 categories:
 - Declaration
 - Scriptlet
 - Expression

Declaration

- Used to declare variables and define methods.
- E.g.

<% !

```
int x = 100;  
public void myMethod() {  
    //Some Code  
}
```

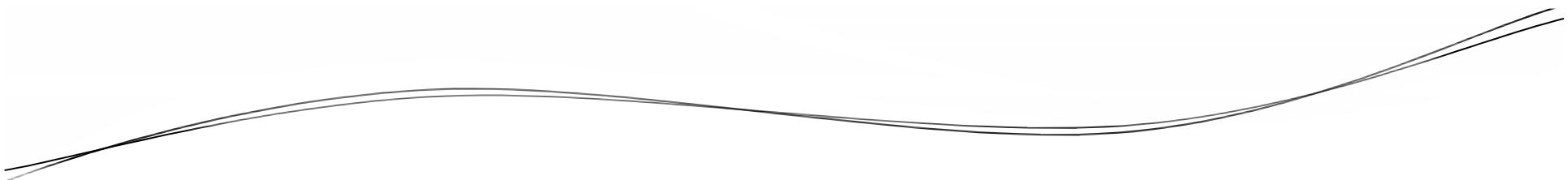
%>

Scriptlet

- Used to write any valid Java code.
- E.g.

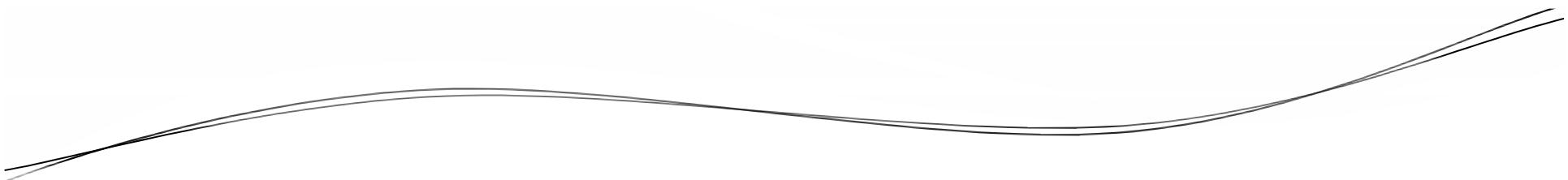
```
<%  
    // Java Statements  
%>
```

- Statements written inside scriptlet execute inside the service method of the servlet.
- Not possible to define methods inside scriptlet.

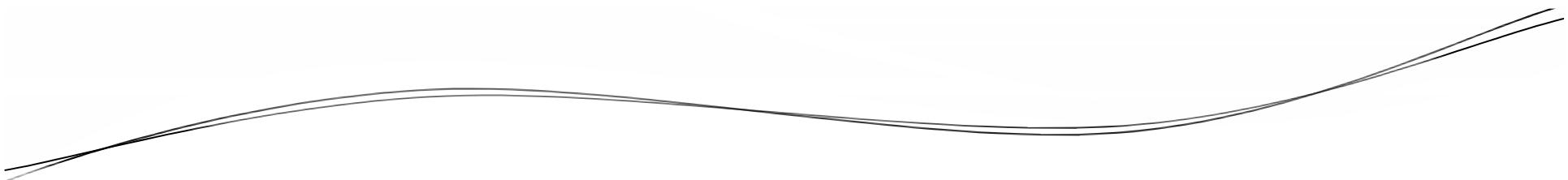


Expressions

- Used to extract value of the variable.
- Expressions are directly processed on the browser window.
- E.g.
`<%=<expr>%>`
- Methods returning ‘void’ cannot be invoked using expressions.



Implicit Objects



Implicit Objects

- request
- response
- out
- session
- config
- application
- page
- pageContext
- exception

Implicit Objects

- request
 - An object of type `javax.servlet.http.HttpServletRequest` that can be used to handle request specific parameters.
 - E.g.

```
<%  
    String str =  
        request.getParameter("name");  
%>
```

Implicit Objects

- response
 - An object of type `javax.servlet.http.HttpServletResponse` that can be used to handle response.
 - E.g.

<%

```
String url =  
    response.encodeURL("hello");
```

%>

Implicit Objects

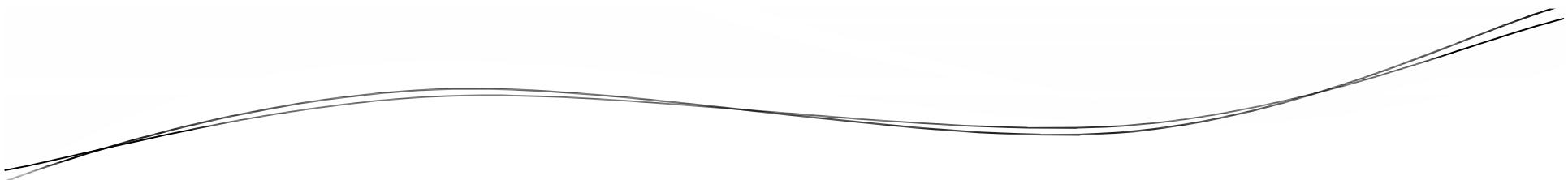
- out
 - An object of type `javax.servlet.jsp.JspWriter` that can be used to generate response through scriptlets.
 - E.g.

```
<%  
    String str = "Welcome";  
    out.println("<h1>" +str+ "</h1>");  
%>
```

Implicit Objects

- session
 - An object of type `javax.servlet.http.HttpSession` that can be used to handle the information at the session level.
 - E.g.

```
<%  
    session.setAttribute  
        ("name", "James");  
%>
```



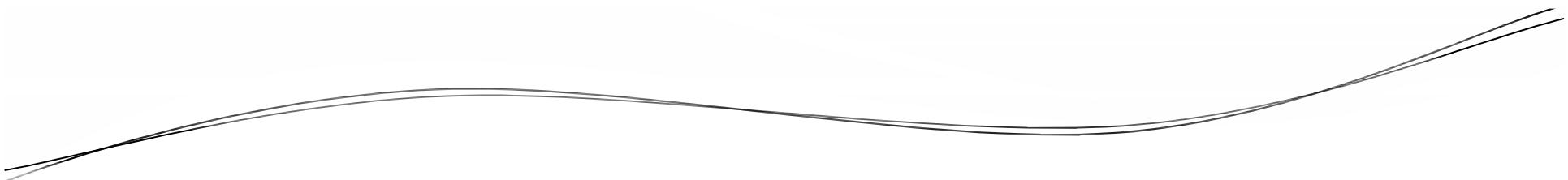
Implicit Objects

- config
 - An object of type
`javax.servlet.ServletConfig`

Implicit Objects

- application
 - An object of type `javax.servlet.ServletContext` that can be used to handle application level information.
 - E.g.

```
<%\n        application.setAttribute\n        ("message", "Welcome");\n%>
```



Implicit Objects

- page
 - It is a reference that refers to an object of JSP specific servlet implementation class generated by the web container.

Implicit Objects

- pageContext
 - An object of type `javax.servlet.jsp.PageContext` that acts as an SPOC through which it is possible to obtain other resources..

Implicit Objects

- pageContext

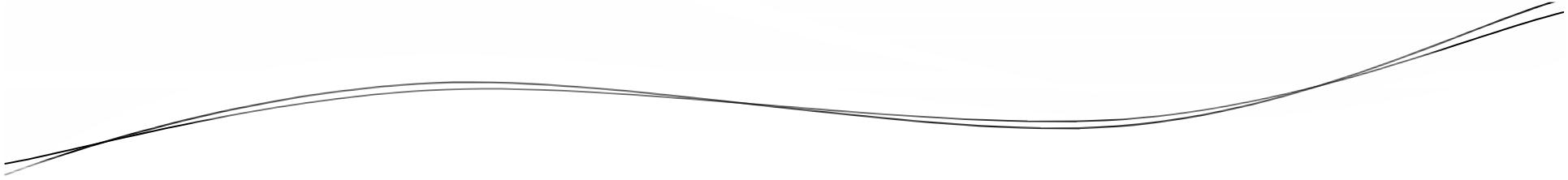
- E.g.

```
<%  
pageContext.  
setAttribute("a","Hello",  
pageContext.REQUEST_SCOPE);
```

```
pageContext.  
setAttribute("b","Welcome",  
pageContext.SESSION_SCOPE);
```

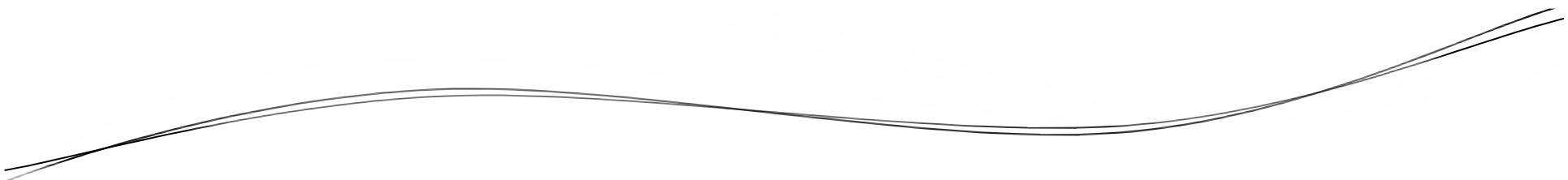
%>

%>



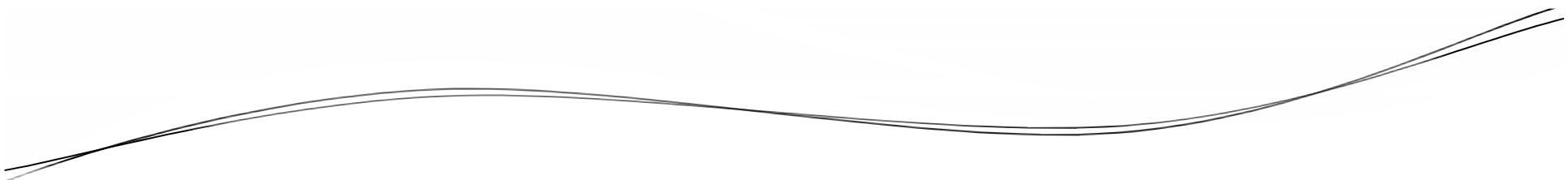
Implicit Objects

- exception
 - An object of type Java.lang.Throwable that can be used to handle exceptions occurred during the execution.
 - It is available only when isErrorPage attribute of page directive is set to true.

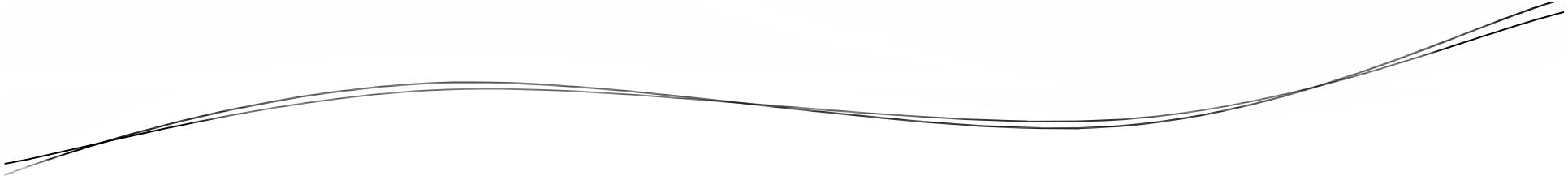


Let's Summarize

- What is JSP, Why JSP
- Life Cycle of JSP
- JSP Tags
- Implicit Objects

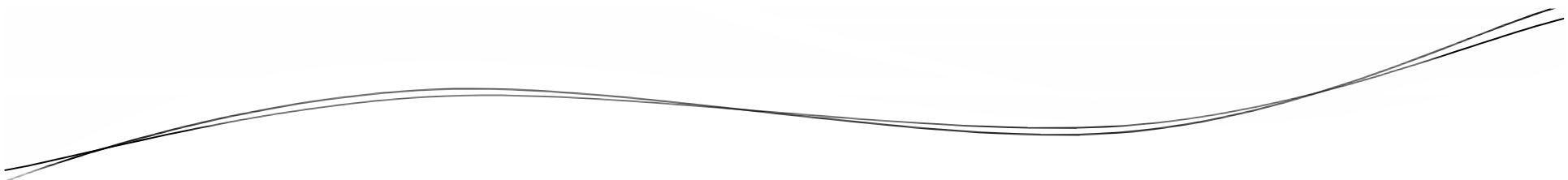


JSP Part 2



Objectives

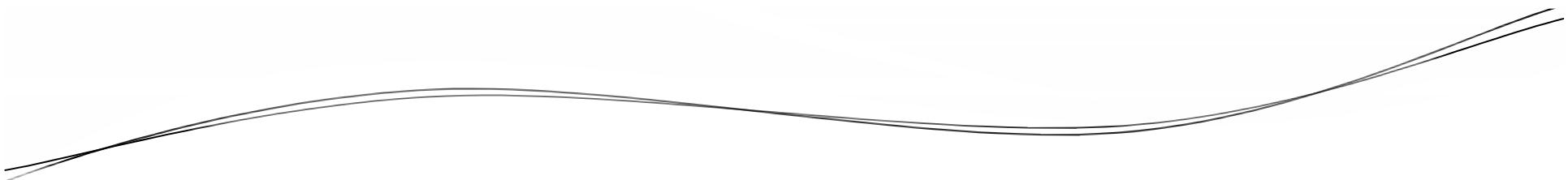
- Understanding Standard Actions in JSP
- Introduction to Custom Tag, and The Need
- Creating Custom Tags



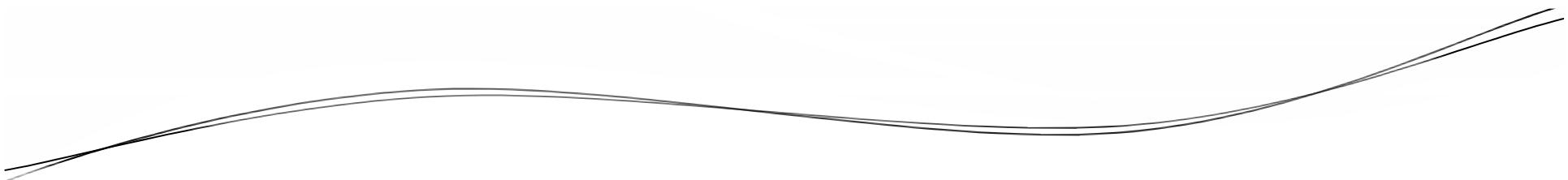
Standard Actions

Standard Actions

- Standard Actions are used to perform some specific task.
- All JSP standard actions follow a standard format:
 - <prefix:suffix>
 - suffix is the actual name of the tag.



<jsp:useBean>



<jsp:useBean>

- Used to instantiate a Java Bean.
- Important Attributes:
 - id
 - class
 - type
 - scope

<jsp:useBean>

- E.g.

```
<jsp:useBean id="d1"  
    class="java.util.Date"  
    scope="session"/>
```

<jsp:setProperty>

- Used to set properties of the bean.
- Syntax:

```
<jsp:setProperty name="<beanName>"  
.....<Property-Details>...../>
```

<jsp:setProperty>

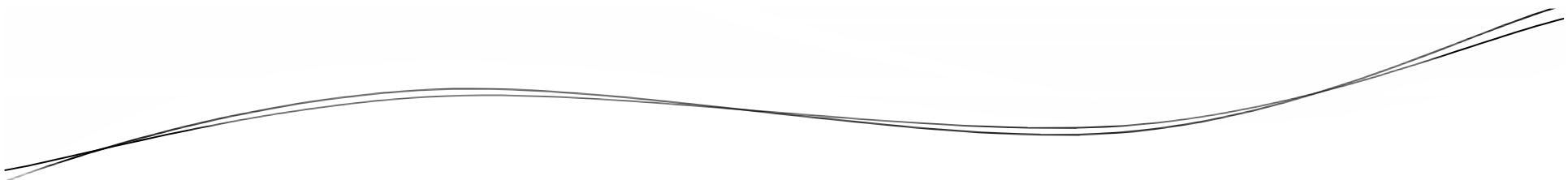
- Property Details:

- property="*"
- property=<property>"
- property=<property>" param=<param>"
- property=<property>" value=<value>"

<jsp:getProperty>

- Used to retrieve properties of the bean.
- Syntax:

```
<jsp:getProperty name="<beanName>"  
                  property="<property>"/>
```



Other Actions

- <jsp:forward>
- <jsp:include>
- <jsp:param>

Other Actions

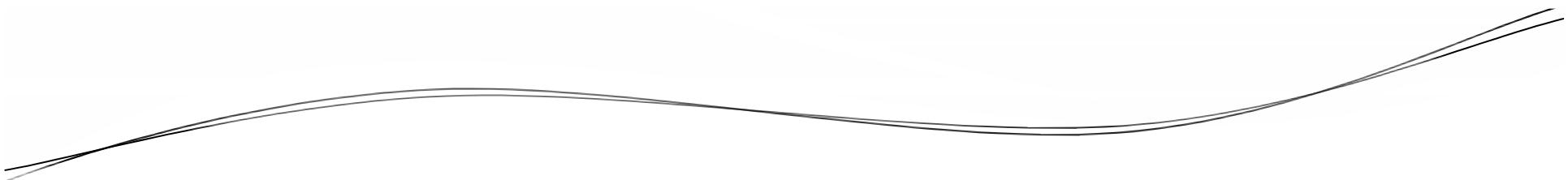
- <jsp:forward>
 - Used to forward the request to another page.
 - E.g.
`<jsp:forward page="next.jsp"/>`

Other Actions

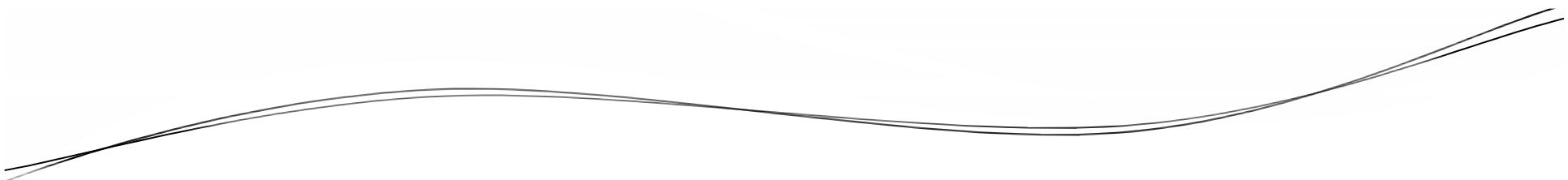
- <jsp:include>
 - Used to include the resources in the existing JSP.
 - E.g.
`<jsp:include page="next.jsp"/>`

Other Actions

- <jsp:param>
 - Used in conjunction with either <forward> or <include>, to supply additional parameters.
 - E.g.
`<jsp:param name=".." value=".."/>`



Custom Tags

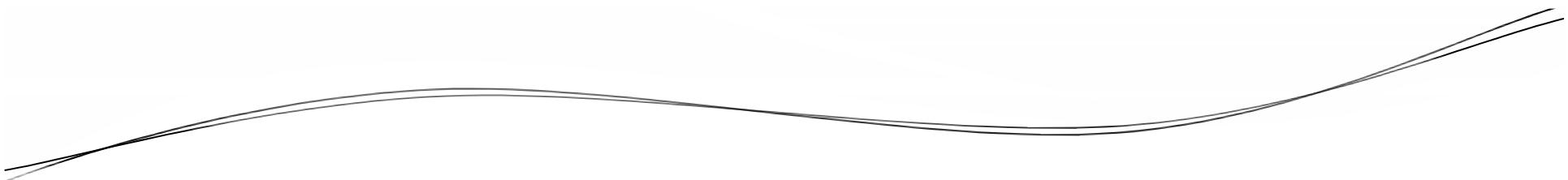


Custom Tags

- Sometimes, as per the application's requirement, it is necessary to create user defined tags.
- Such tags are called as custom tags.

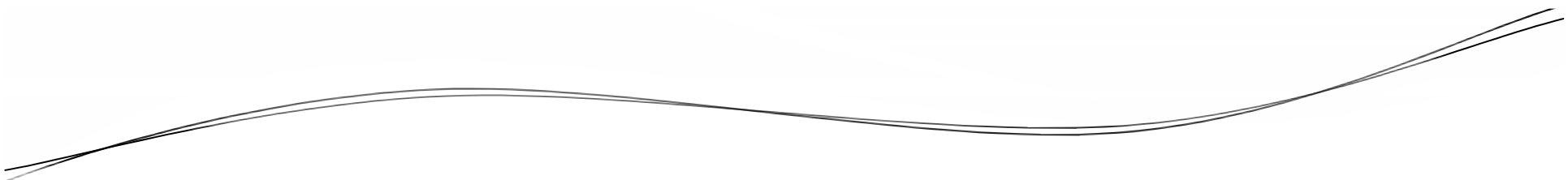
Custom Tags

- Need
 - <jsp:useBean> is capable of working upon only Value Objects.
 - To handle the processing, still developer is required to write a Java code inside scriptlets.



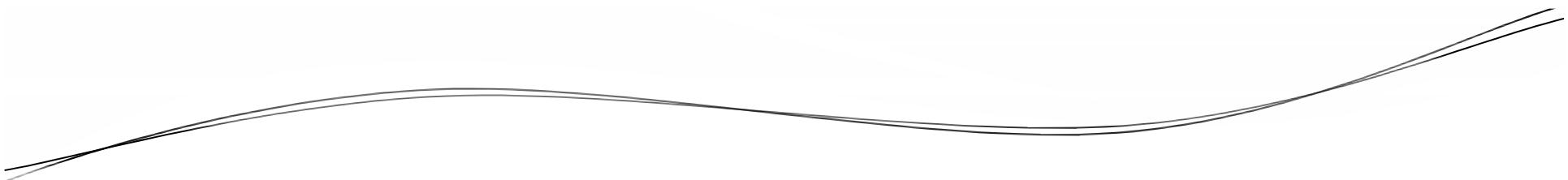
Custom Tags

- Benefits:
 - Custom tags can be used to reduce the no of scriptlets in JSP.
 - Reusability
 - Can be used to handle bean processing.



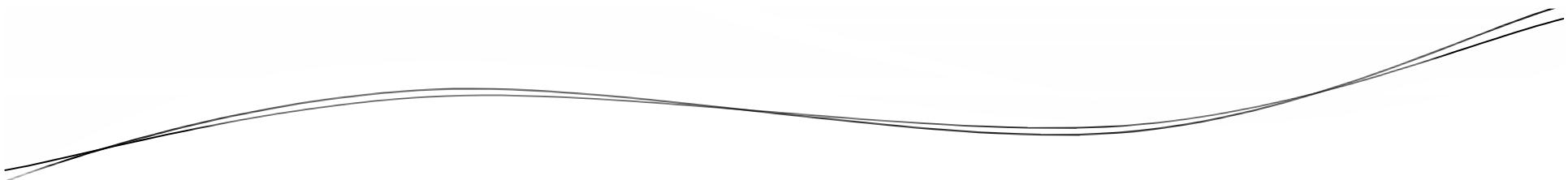
Custom Tags

- To create custom tags, developer has to provide the implementation of the tag using a Java class that is known as a Tag Handler Class.



Custom Tags

- The tag handler class is to be created using a Tag API that belongs to a package `javax.servlet.jsp.tagext`.
- It mainly includes the following:
 - Tag
 - TagSupport



Custom Tags

- Once the tag implementation class is defined, it is to be mapped with the name of the tag which can be accomplished by creating a TLD file.
- It is a mapping file that contains entries in the XML format.

Custom Tags

- TLD Format:

<tag>

<name>...</name>

<tag-class>...</tag-class>

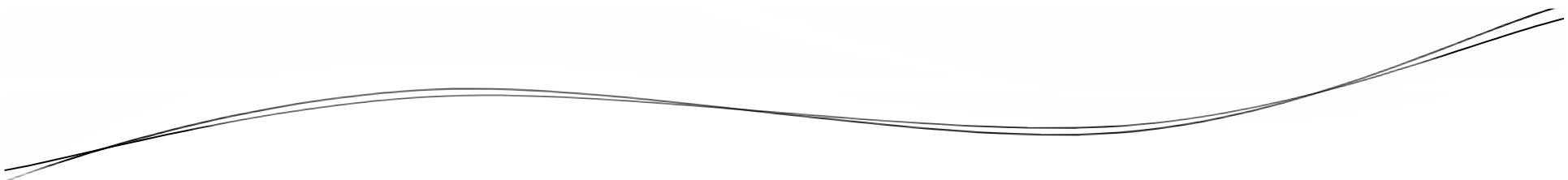
<body-content>...</body-content>

</tag>

Taglib Directive

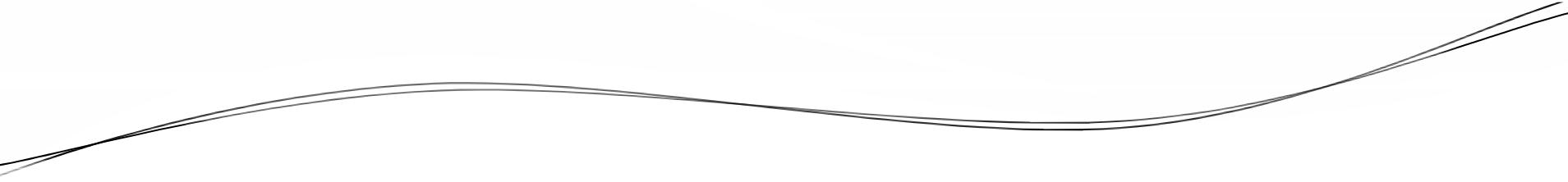
- Used to locate TLD file in the JSP.
- E.g.

```
<%@taglib  
    uri="" prefix=""  
%>
```



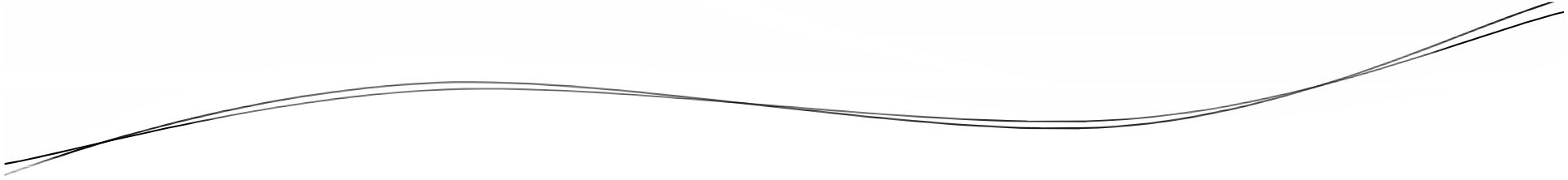
Let's Summarize

- Standard Actions in JSP
- Custom Tag and Their Need



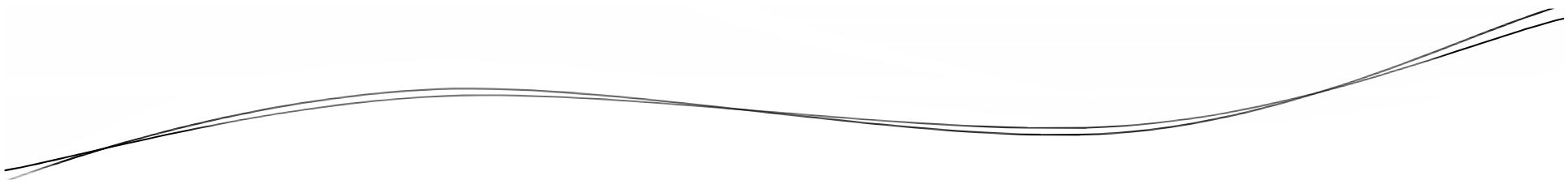
Hibernate

By Rahul Barve



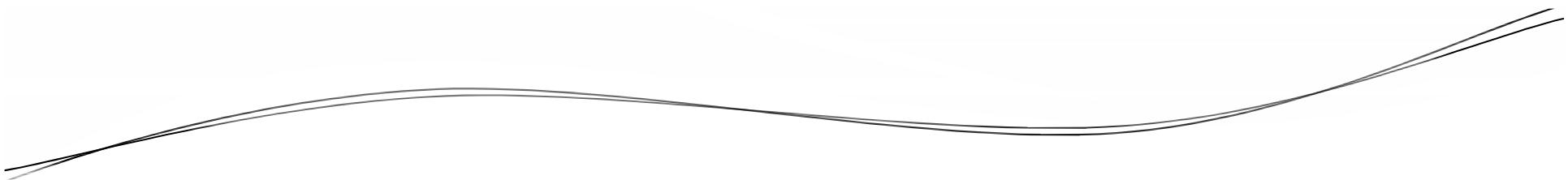
Objectives

- Introduction to Hibernate
- Hibernate Architecture
- First Hibernate Application



Introduction to Hibernate

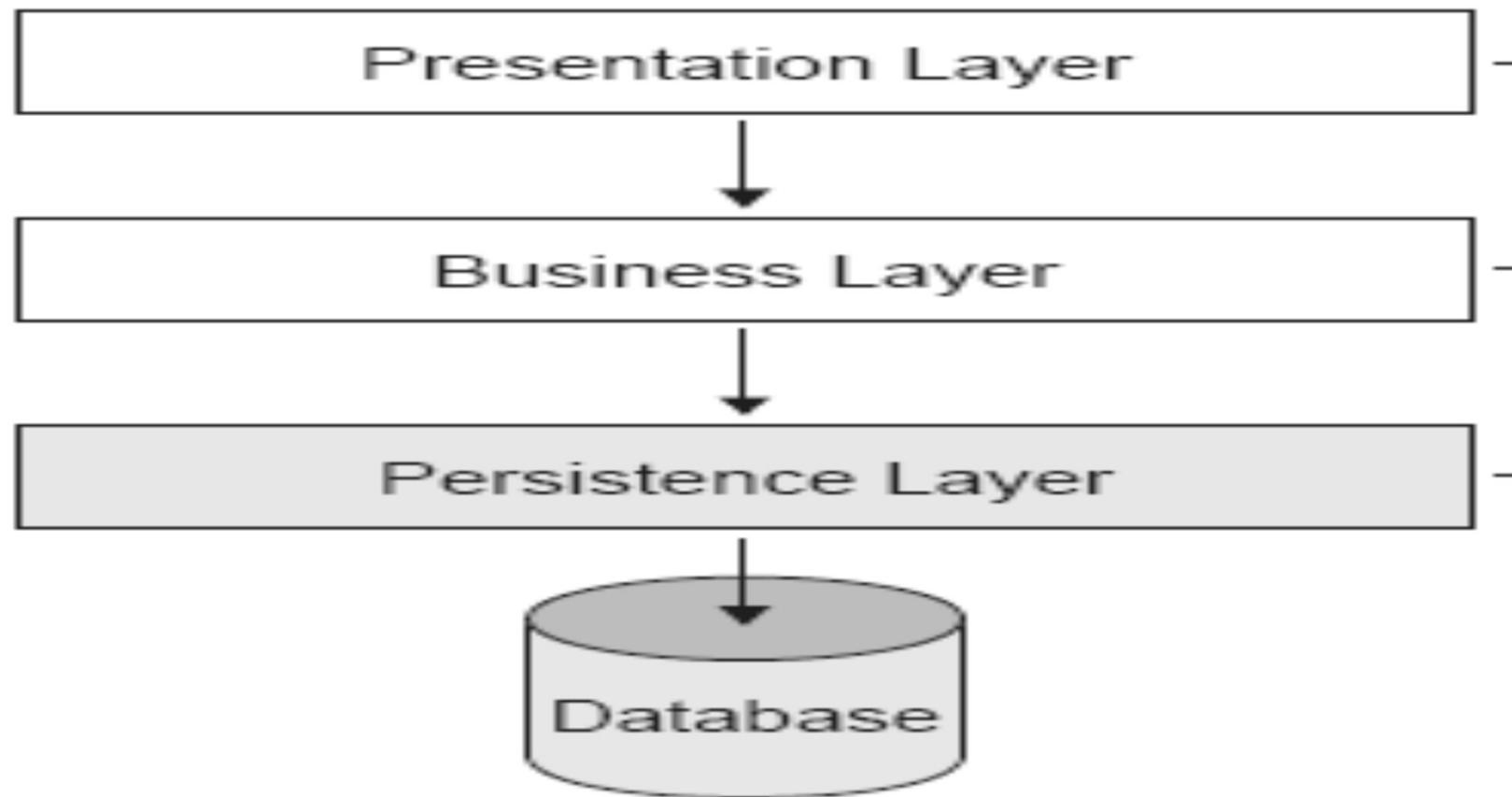
By Rahul Barve



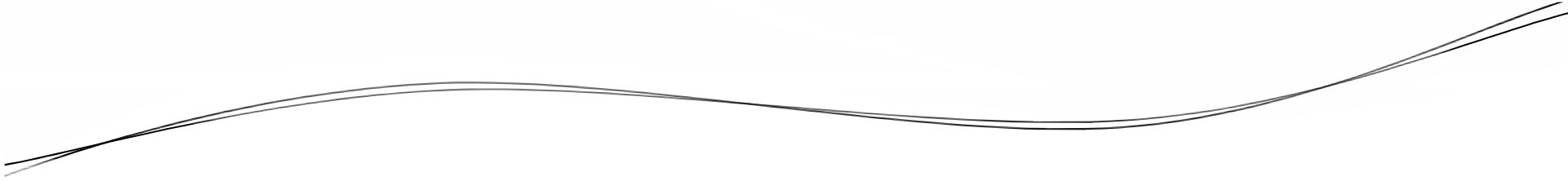
Introduction to Hibernate

- Hibernate is an Open Source Java Based Framework that is used to build a Persistence Layer of an application.

Persistence Layer

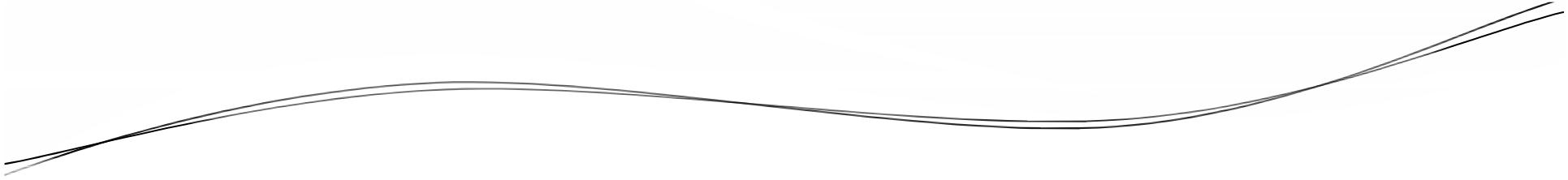


By Rahul Barve



Introduction to Hibernate

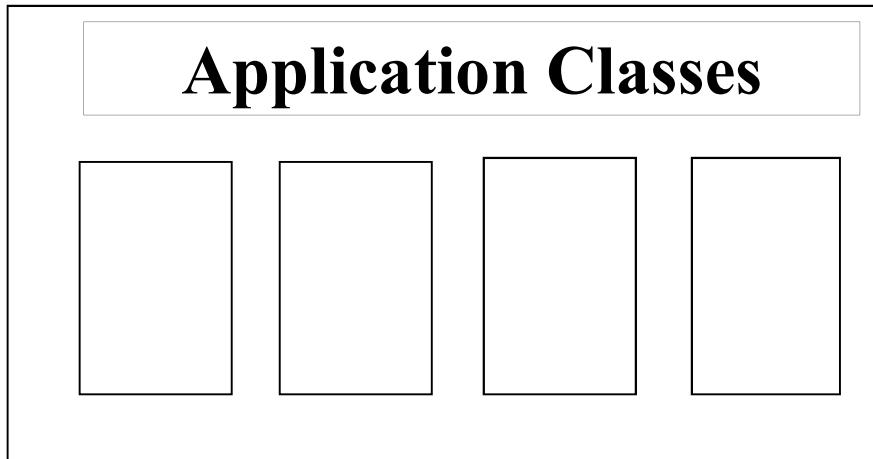
- Hibernate is a framework that is based upon some principles known as ORM.
- It is distributed into a single ZIP file which contains several JAR files.



Hibernate – Basic Architecture

By Rahul Barve

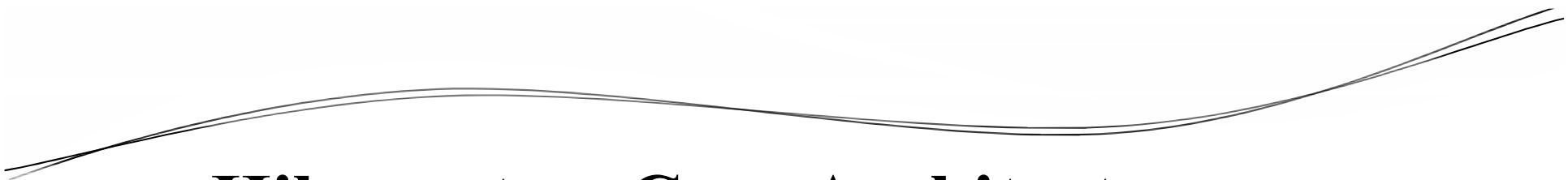
Hibernate – Basic Architecture



Hibernate

Database Server

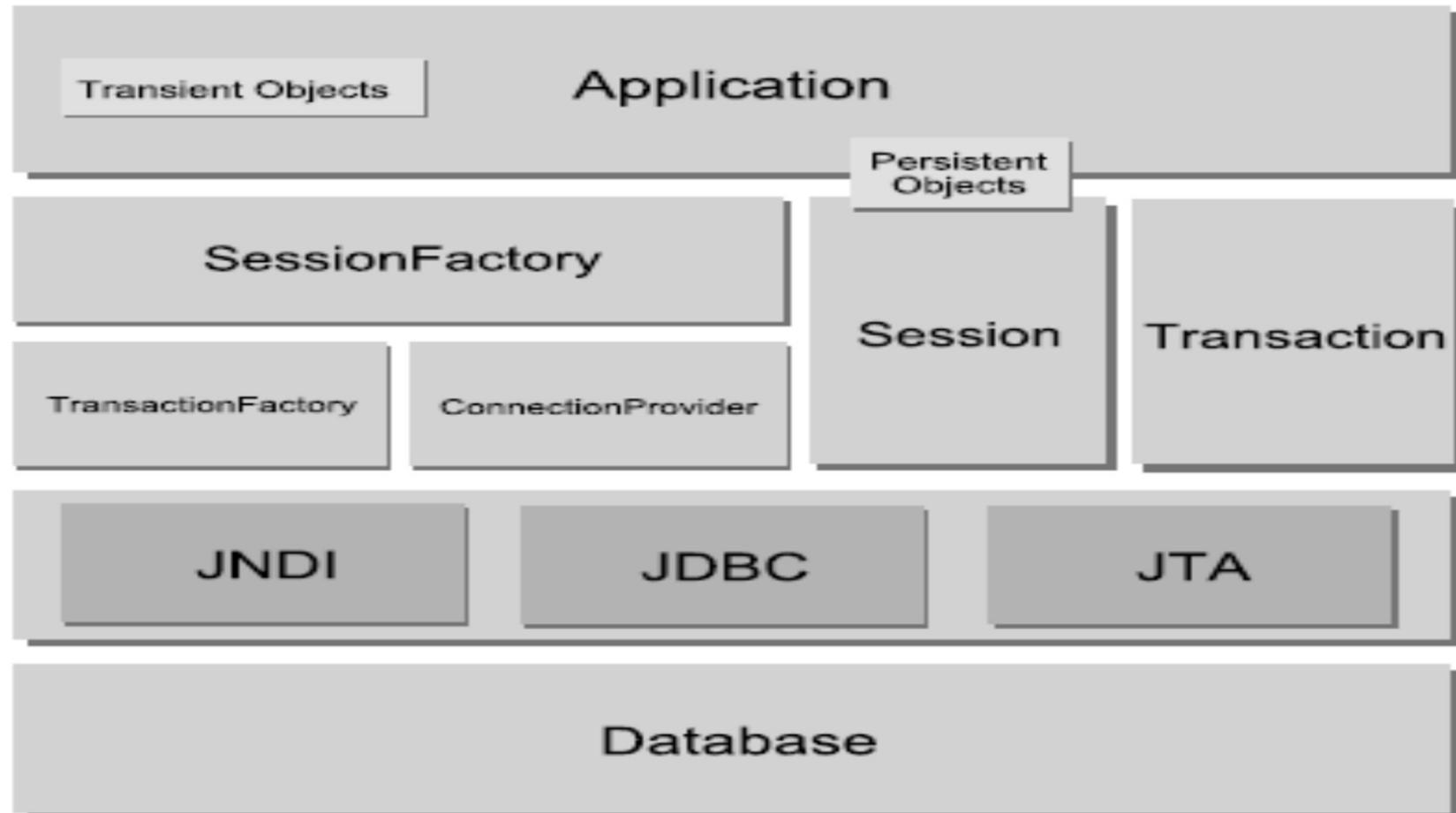
By Rahul Barve



Hibernate – Core Architecture

By Rahul Barve

Hibernate – Core Architecture

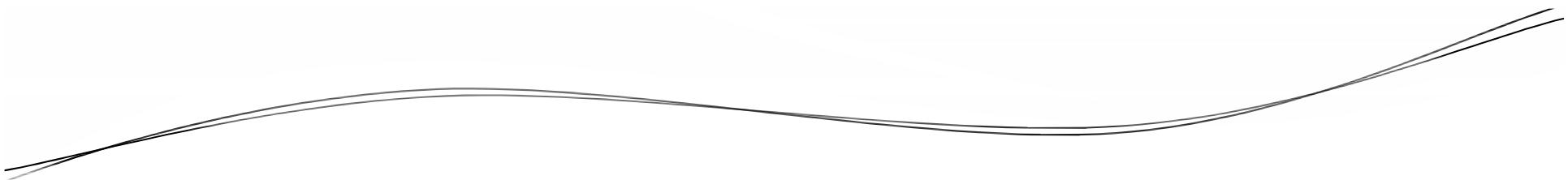


By Rahul Barve



Hibernate – Core API

- Configuration
- Session
- SessionFactory
- Transaction
- Query

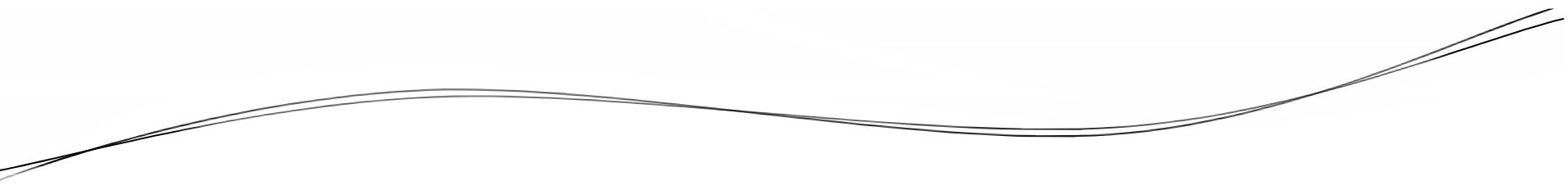


Configuration

By Rahul Barve

Configuration

- org.hibernate.cfg.Configuration
- Used to configure Hibernate based upon the configurations through .properties or .xml files or even programmatically.



Session

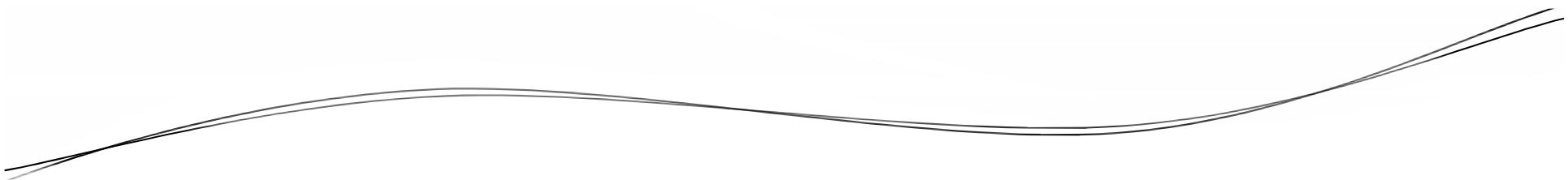
By Rahul Barve

Session

- org.hibernate.Session
- An object representing a conversation between the application and the persistent store.
- Wraps a JDBC connection.
- A client of TransactionFactory.

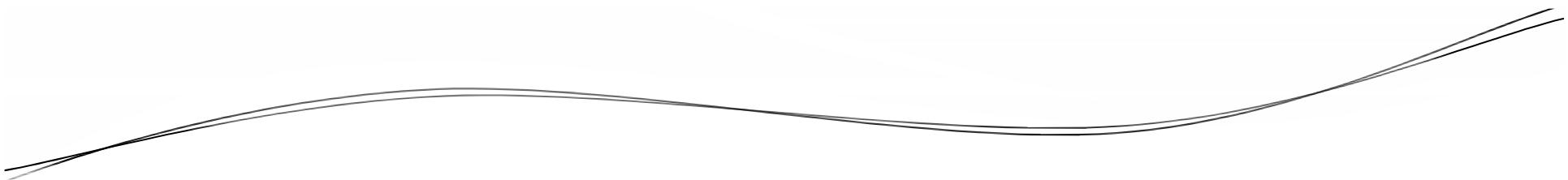
Session

- A Component used to persist or load objects to/from the database.
- Used to perform basic CRUD operations.
- Provides utility methods like `save()` , `update()` , `delete()` , `load()` .



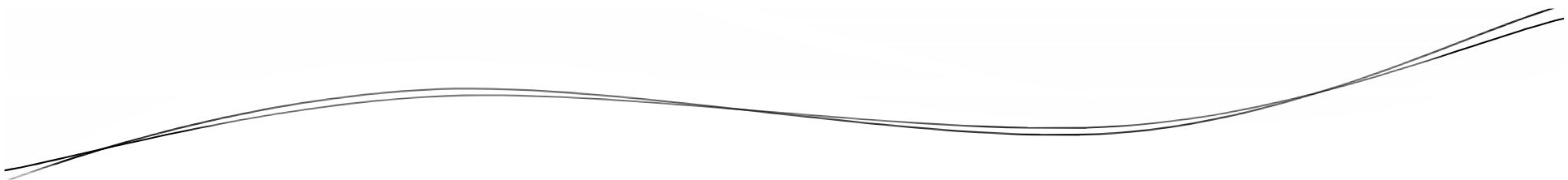
SessionFactory

By Rahul Barve



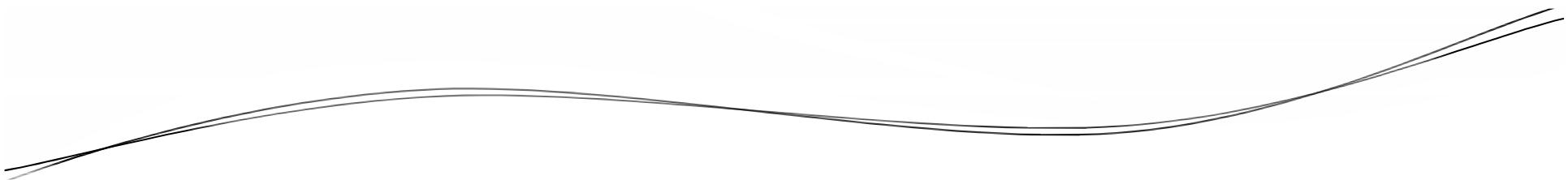
SessionFactory

- org.hibernate.SessionFactory
- A factory for Session and a client of ConnectionProvider.



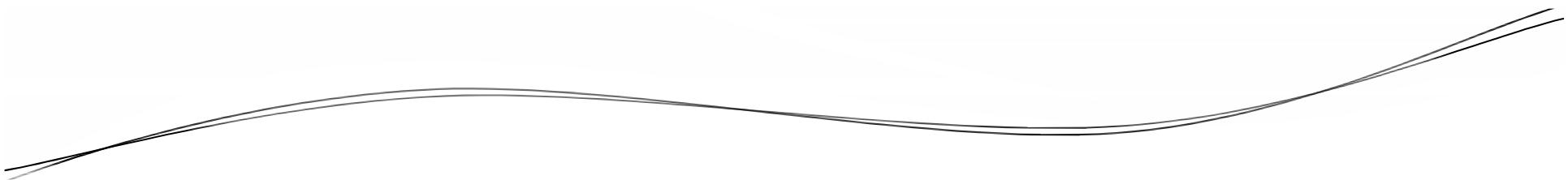
SessionFactory

- Allows to instantiate Session
- Heavyweight Component
- Only one per application



Transaction

By Rahul Barve

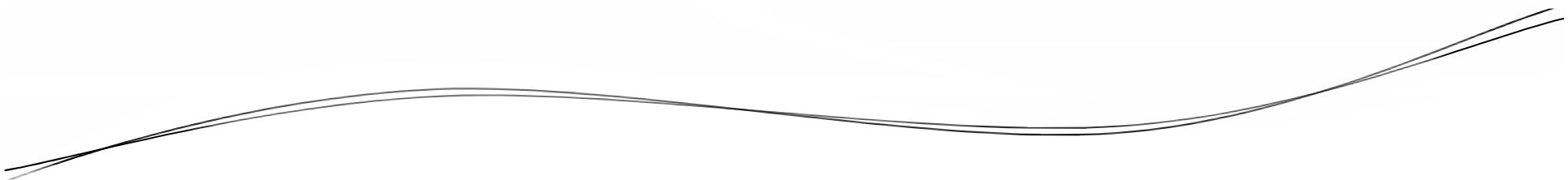


Transaction

- org.hibernate.Transaction
- An object used by the application to specify atomic units of work.

Transaction

- Abstracts application from underlying JDBC or JTA transaction.
- Ensures that all the operations on persisted objects occur in a transaction supported by either JDBC or JTA.
- Maintains Atomicity.

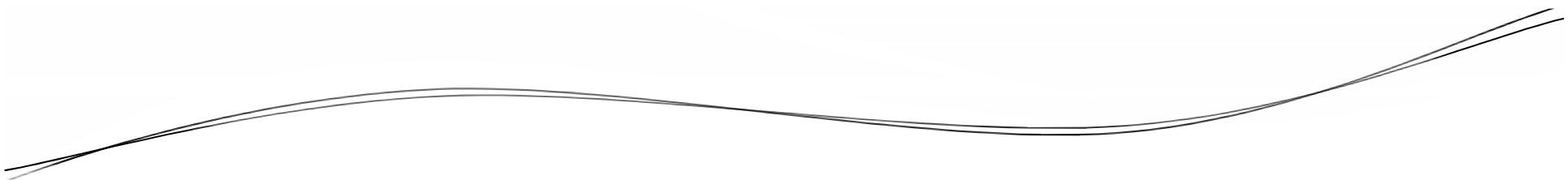


ConnectionProvider

By Rahul Barve

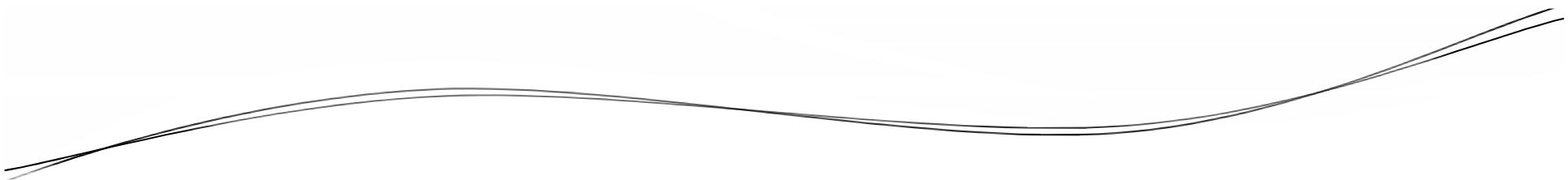
ConnectionProvider

- org.hibernate.connection.ConnectionProvider
- A factory for JDBC connections.
- Abstracts application from underlying Datasource or DriverManager.



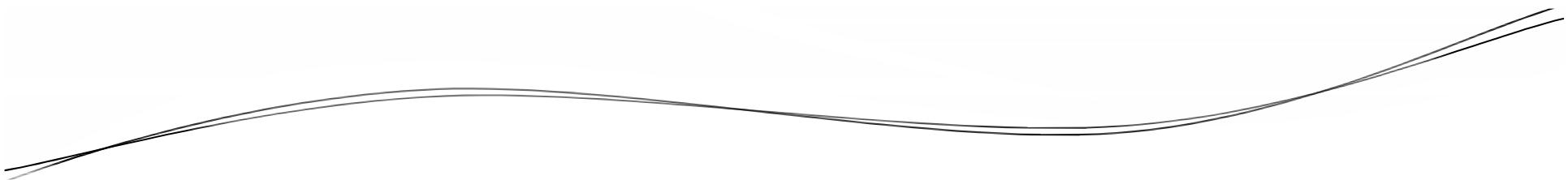
TransactionFactory

By Rahul Barve



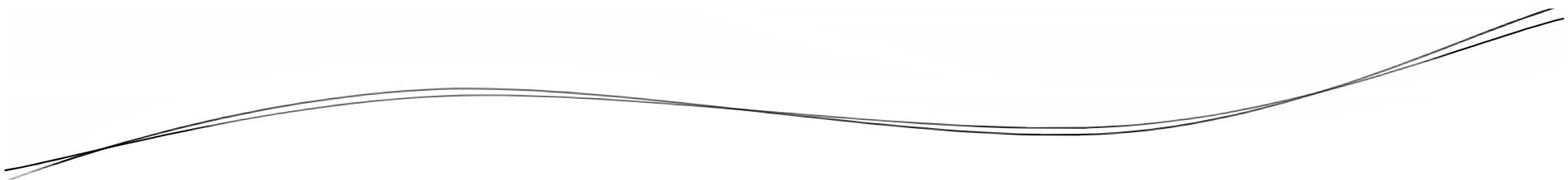
TransactionFactory

- org.hibernate.TransactionFactory
- A factory for Transaction instances.



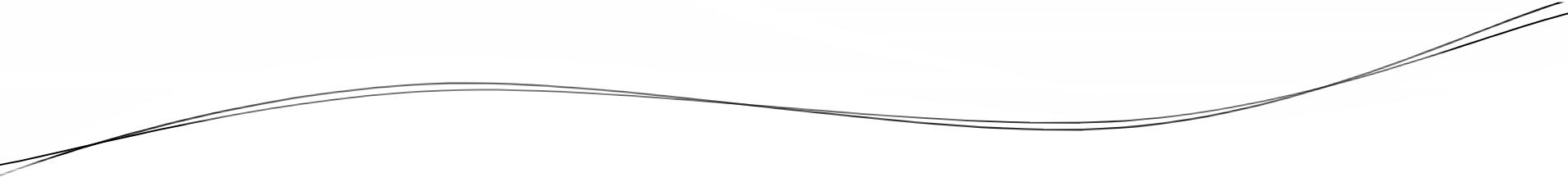
Query

By Rahul Barve

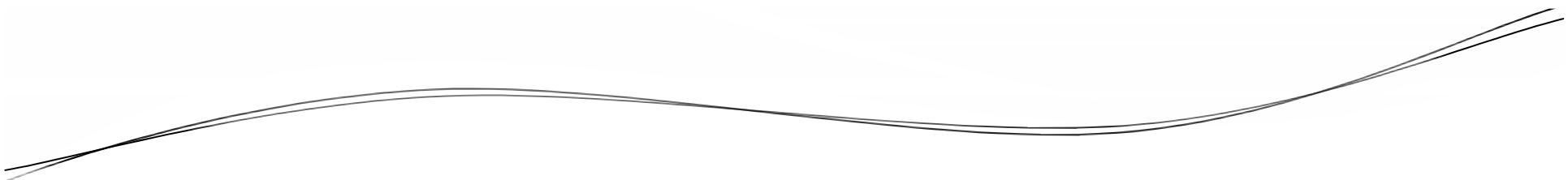


Query

- Used to perform Query operations across the database.
- Uses various clauses, functions to fine tune the query results.

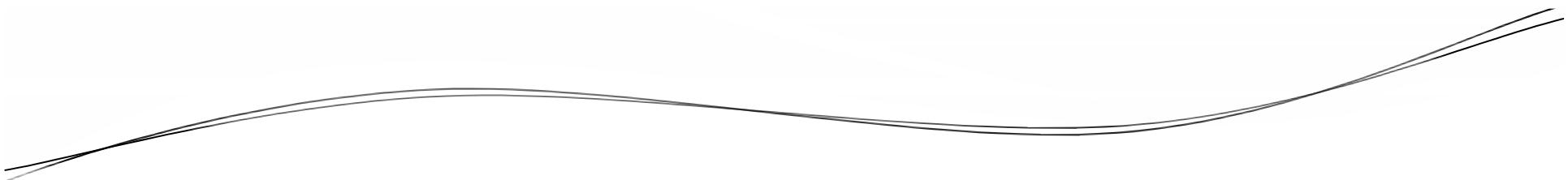


Hibernate with Annotations

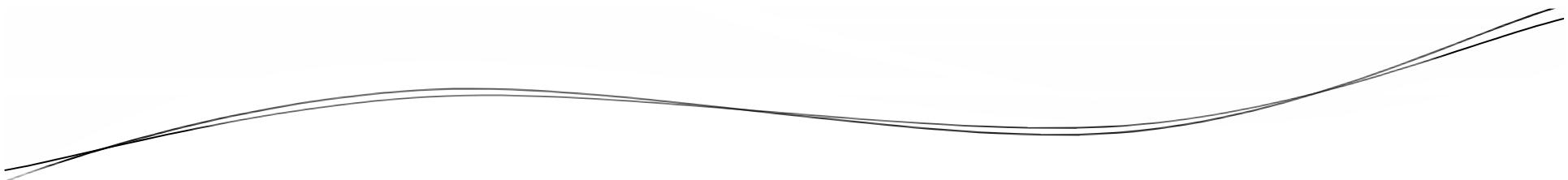


Objectives

- Introduction to Hibernate Annotations
- Introduction to JPA
- Working with Annotations
- A Simple Example

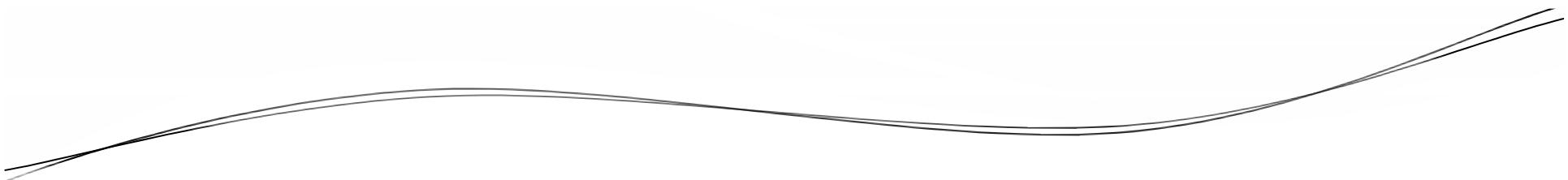


Hibernate Annotations



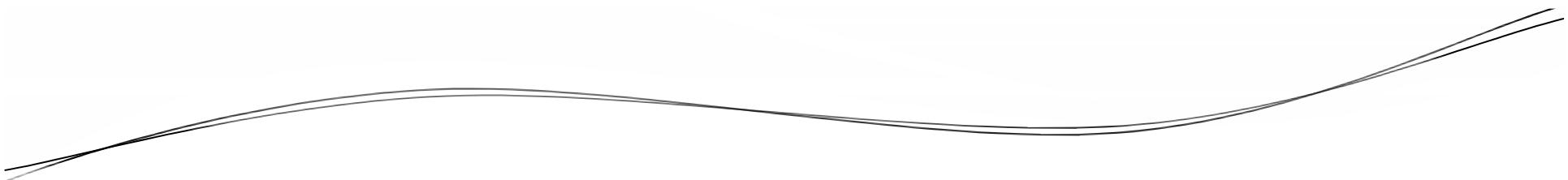
Hibernate Annotations

- Hibernate provides support for Mapping between Class and Table, or Field and Column using Annotations

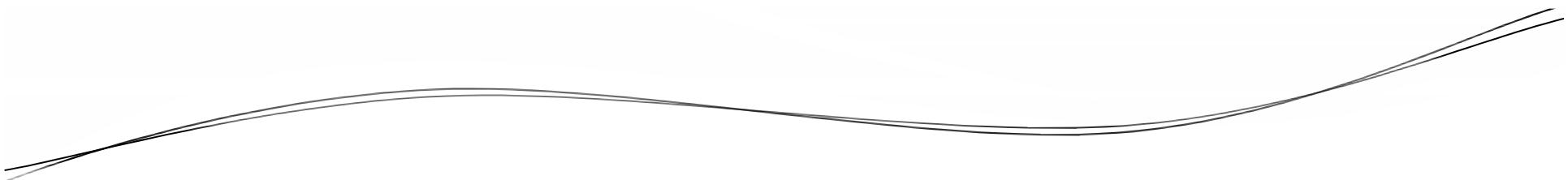


Hibernate Annotations

- There are 2 ways to handle Annotation Based mapping:
 - Using Hibernate Annotations
 - Using JPA Annotations

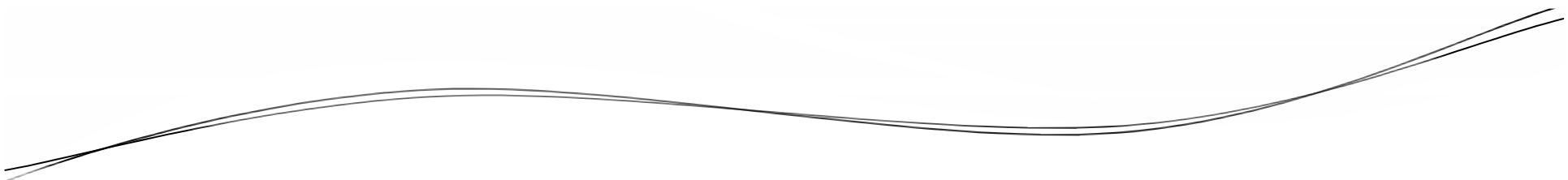


What is JPA



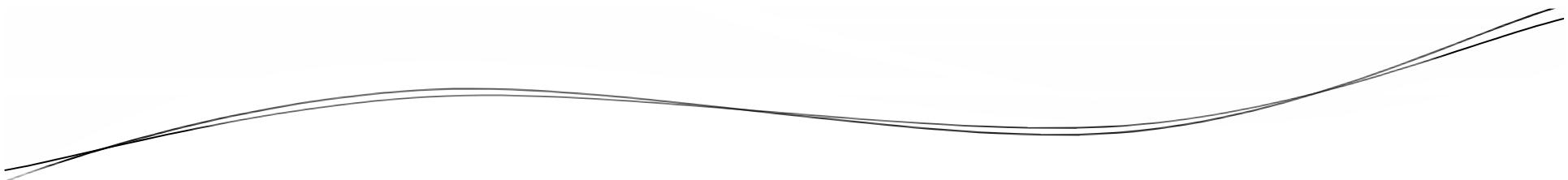
What is JPA

- JPA stands for Java Persistence API.
- A specification for managing entities in the application domain.

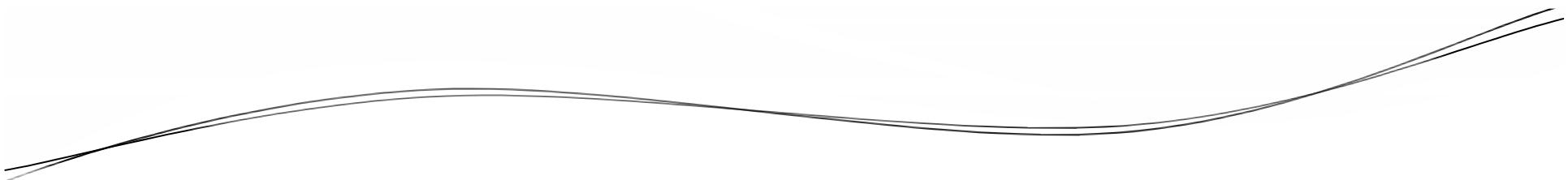


What is JPA

- It is an abstraction on the top of any ORM framework like Hibernate, Toplink, Eclipselink, Ibatis and so on.

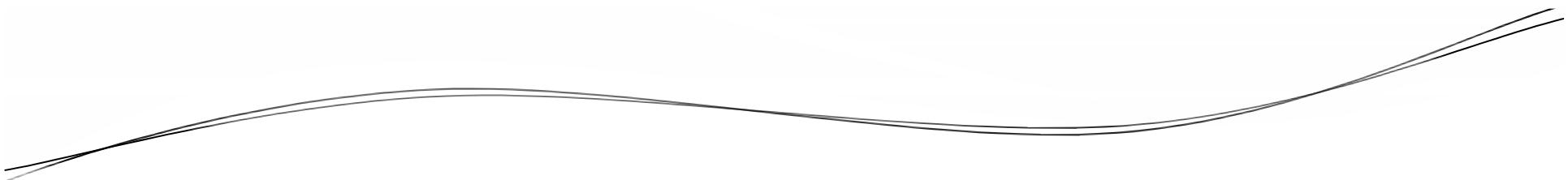


Why JPA



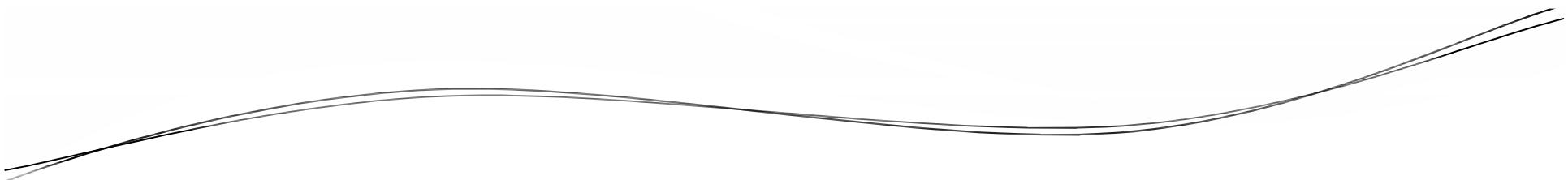
Why JPA

- Provides ease-of-use abstraction on top of JDBC.
- Application code can be isolated from database, vendor specific peculiarities and optimization.

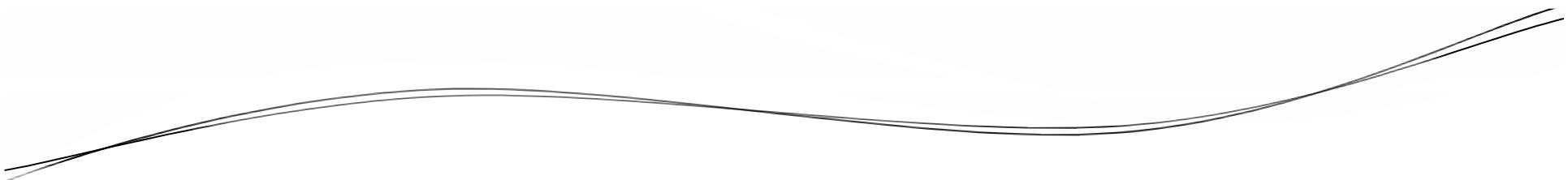


Why JPA

- An Object-to-Relational Mapping (ORM) engine.
- Provides a query language that is tailored to work with java objects rather than relational schema.

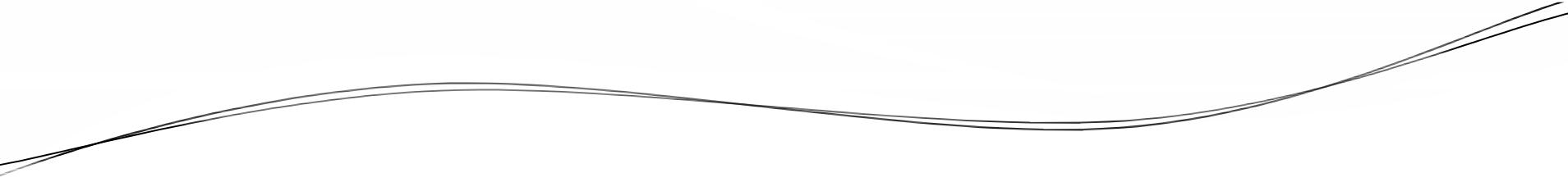


JPA Entities



JPA Entities

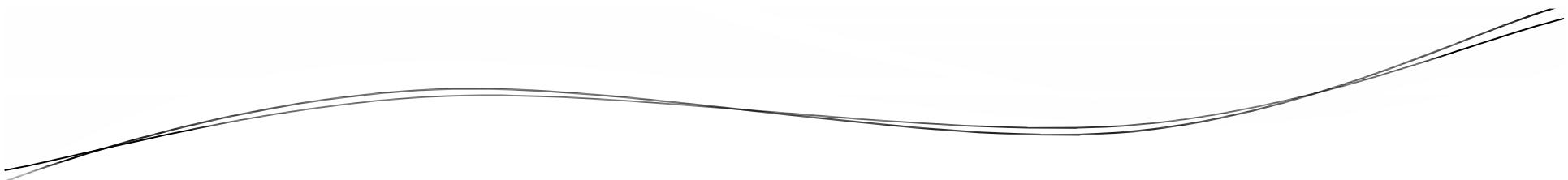
- JPA manages persistency with the help of simple Java Entities.
- JPA Entities are simple POJOs.



Basic Annotations

Basic Annotations

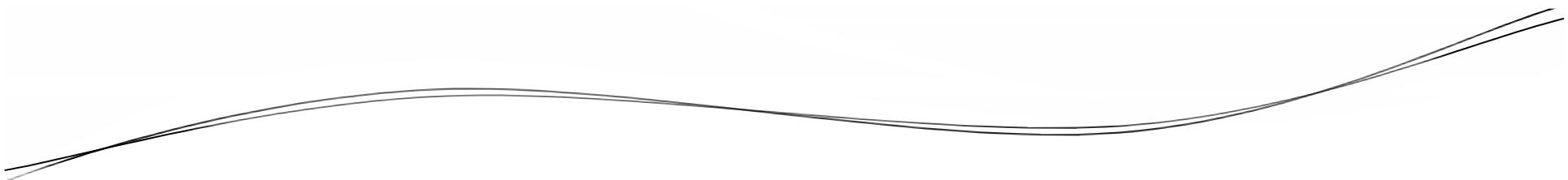
- JPA provides 4 basic annotations:
 - `@Entity`
 - `@Table`
 - `@Id`
 - `@Column`



@Entity

@Entity

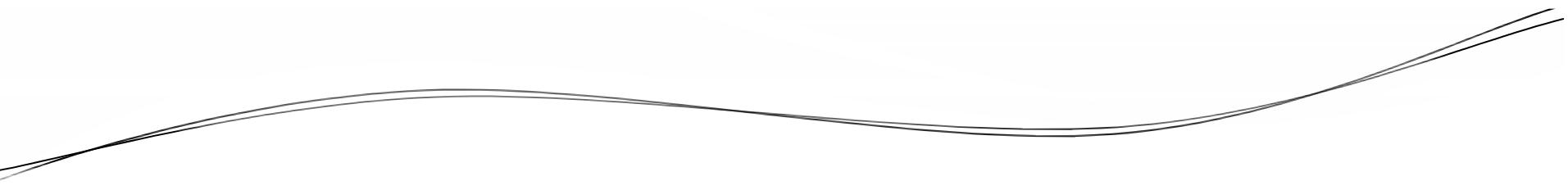
- Tells the persistence provider about the entity class which is being mapped to the database table.
- Mandatory as far as annotation based metadata is concerned.



@Table

@Table

- Tells the EntityManager service about the relational table to which the bean class maps.
- If omitted, it defaults to the unqualified name of the entity class.



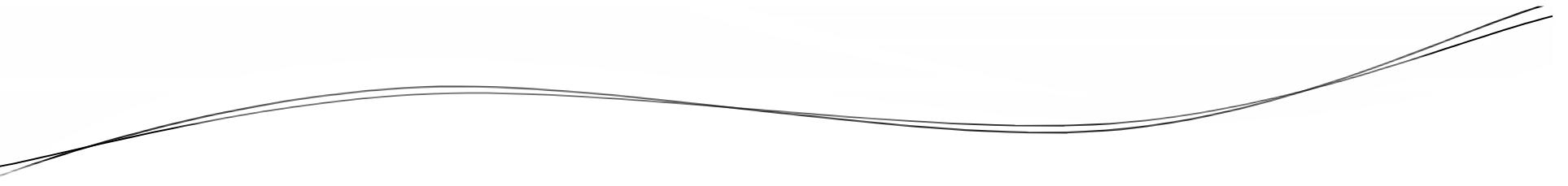
@Column

By Rahul Barve

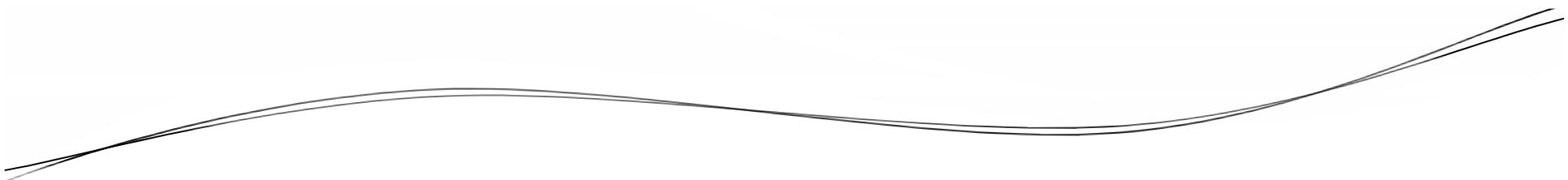
20

@Column

- Specifies the respective DB column to which the entity class field maps.
- Not required if class field name matches with the DB column name.

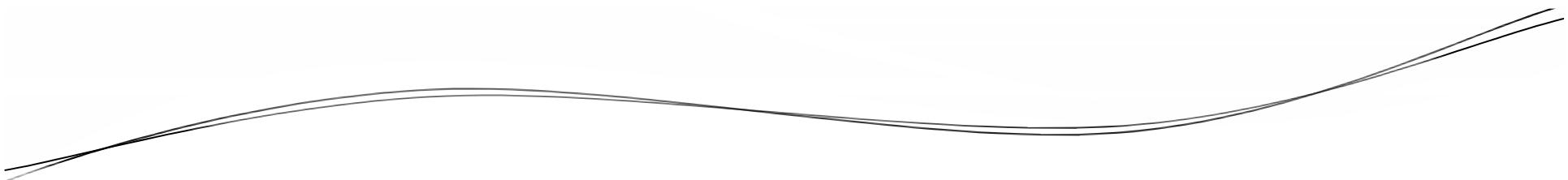


@Id



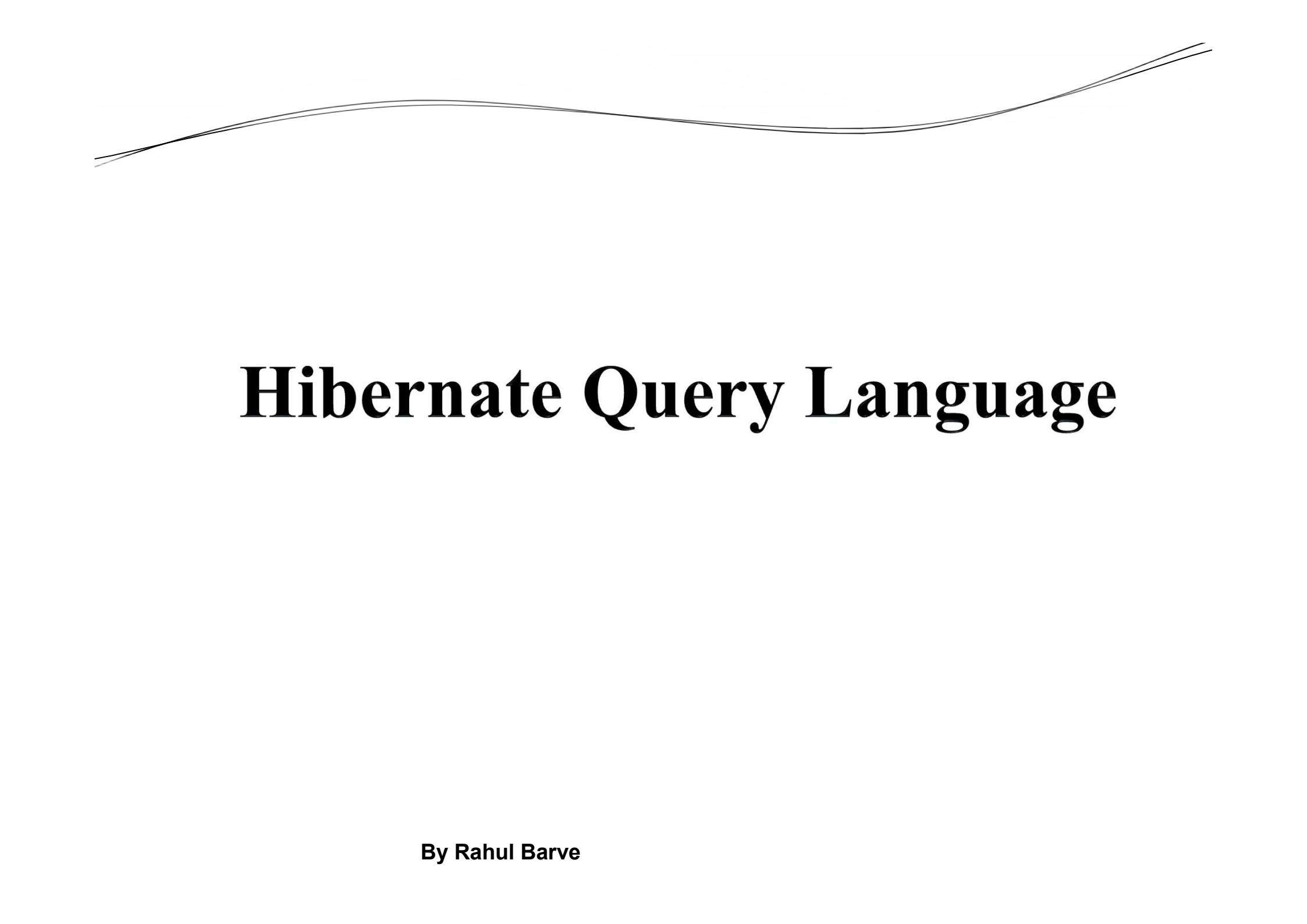
@Id

- Identifies one or more properties that make up the primary key for the table.
- Mandatory as far as annotation based metadata is concerned.



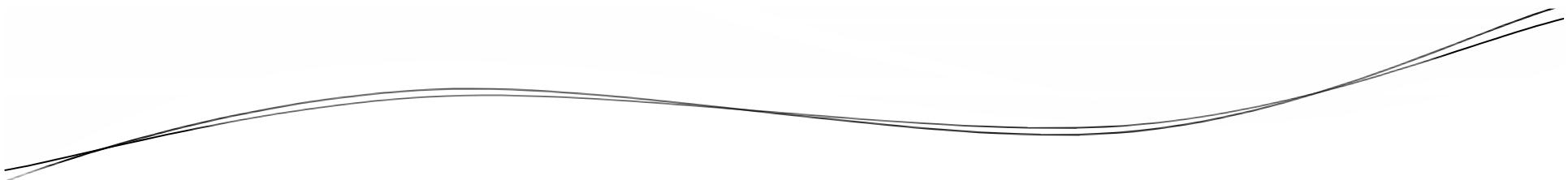
Let's Summarize

- Introduction to Hibernate Annotations
- Introduction to JPA
- Working with Annotations
- A Simple Example



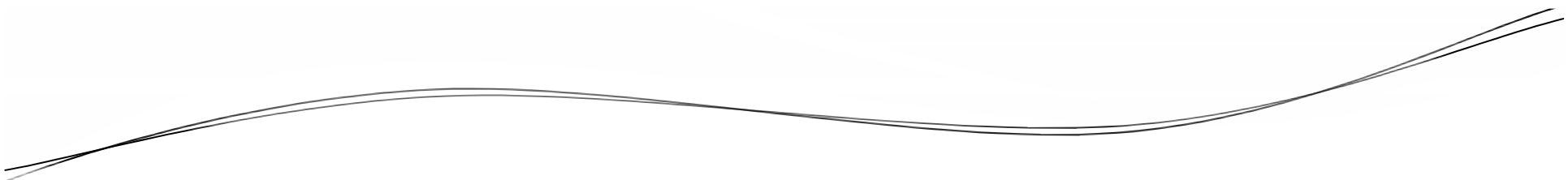
Hibernate Query Language

By Rahul Barve



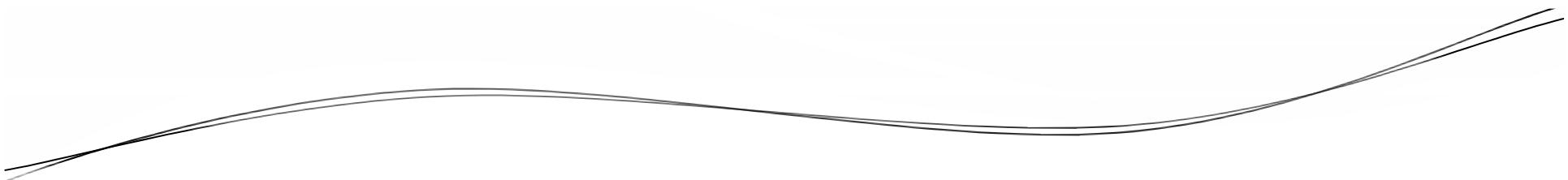
Objectives

- Describe Hibernate Query Language (HQL)
- Working with Queries



Features

By Rahul Barve

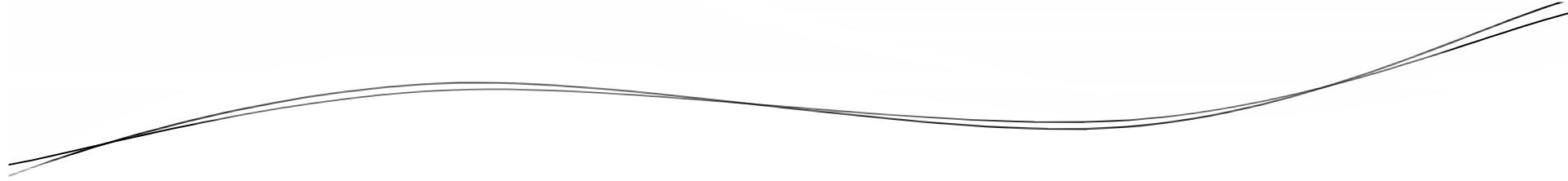


Features

- Used to execute queries against database.
- Based on the relational object models.

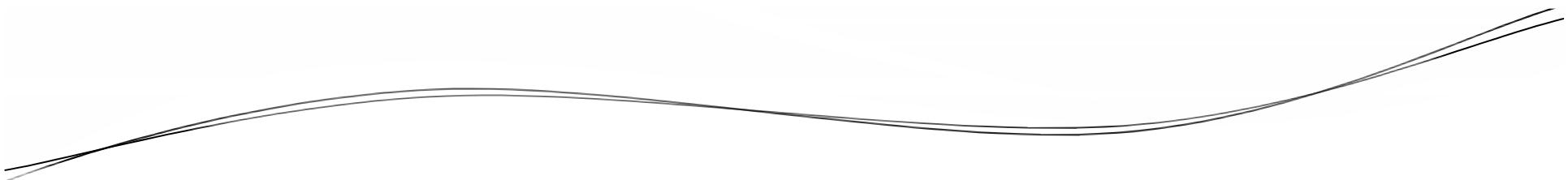
Features

- Uses Classes and properties instead of tables and columns.
- Extremely powerful and it supports Polymorphism and Associations.



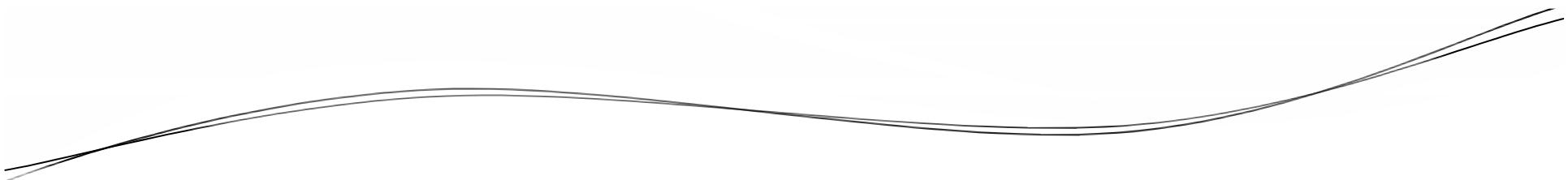
Features

- The ability to apply restrictions to properties of associated objects related by reference or held in collections.
- The ability to retrieve only properties of an entity or entities, without the overhead of loading the entity.



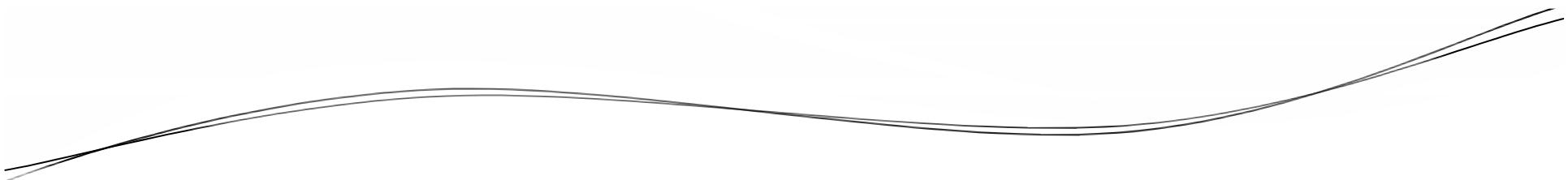
Features

- The ability to order the results of the query.
- The ability to paginate the results.



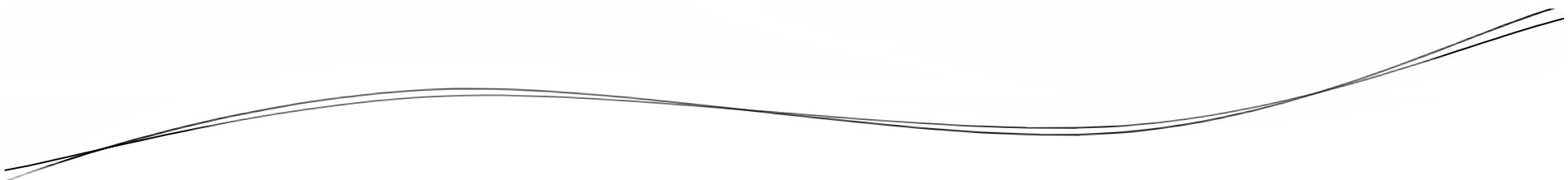
Features

- Provides support for aggregation with group by, having, and aggregate functions like sum, min and max.
- Outer joins when retrieving multiple objects per row.



Clauses

By Rahul Barve



Clauses

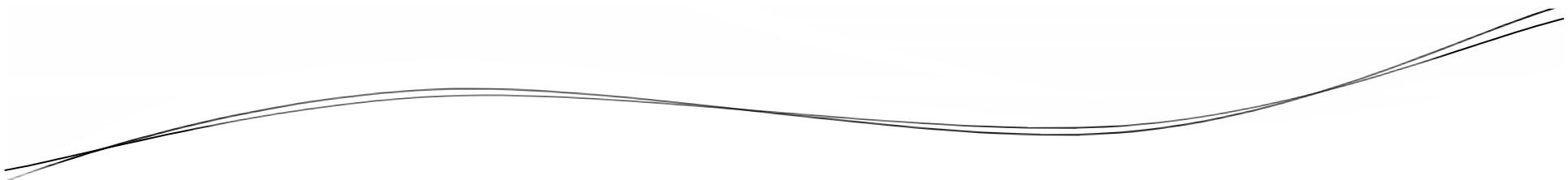
- FROM
- SELECT
- WHERE
- ORDER BY

FROM

- Used to load a set of entities from the underlying database table.
- The returned collection contains instances of an entity class.
- General Form: `from Employee e`

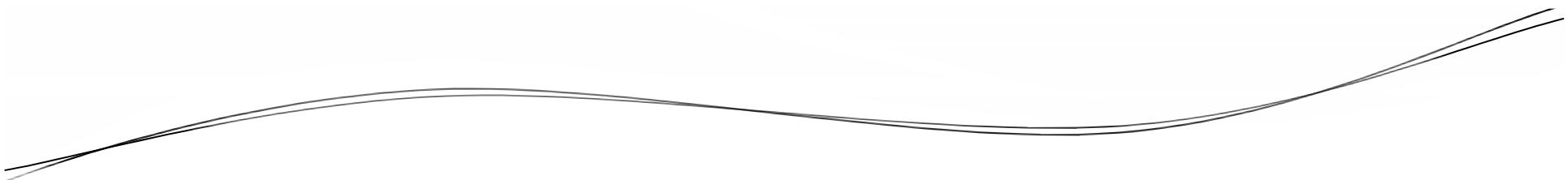
SELECT

- Used to retrieve selected fields from the underlying database table.
- The returned collection contains instances of array type.
- General Form: select e.name, e.sal from Employee e

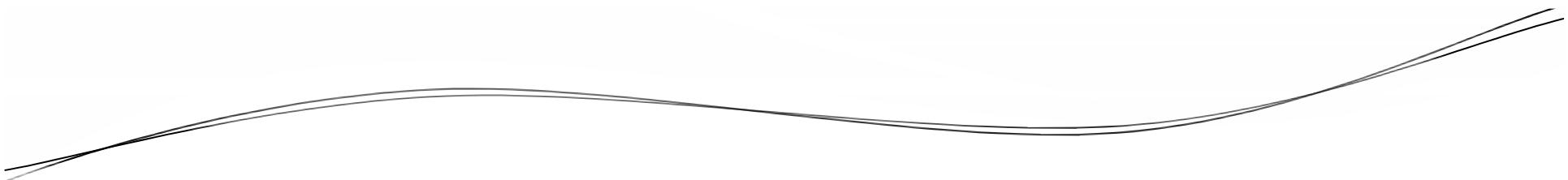


Let's Summarize

- HQL
- Querying using HQL

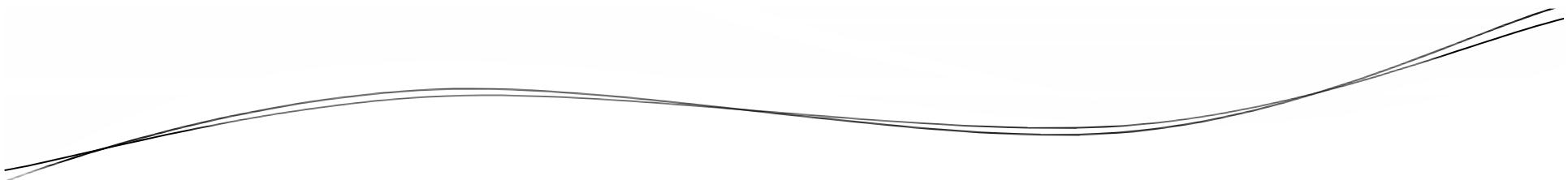


Introduction to Spring

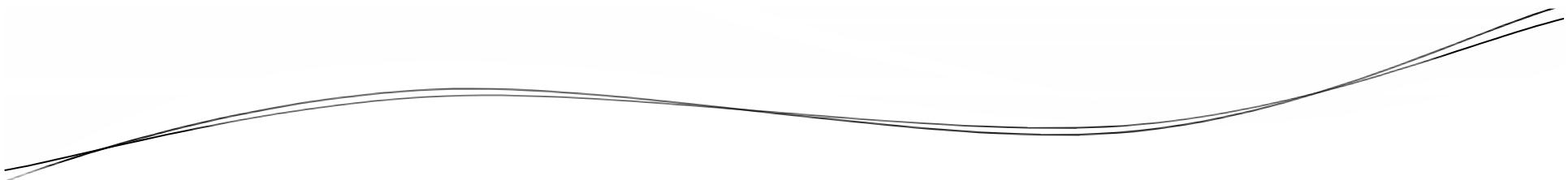


Objectives

- What is Spring
- Why Spring
- Spring Features
- Spring Modules
- Spring Architecture
- First Example

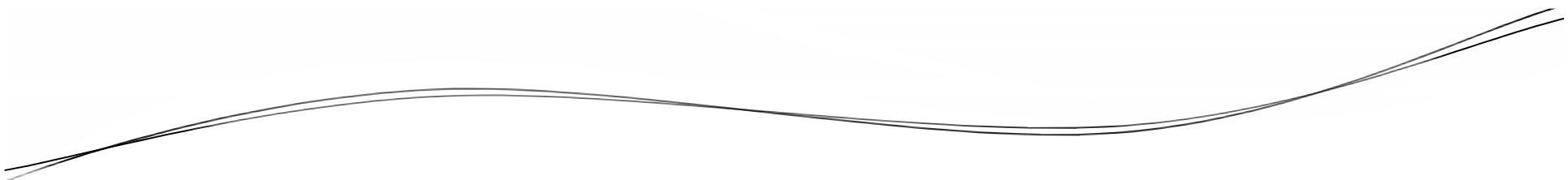


What is Spring?



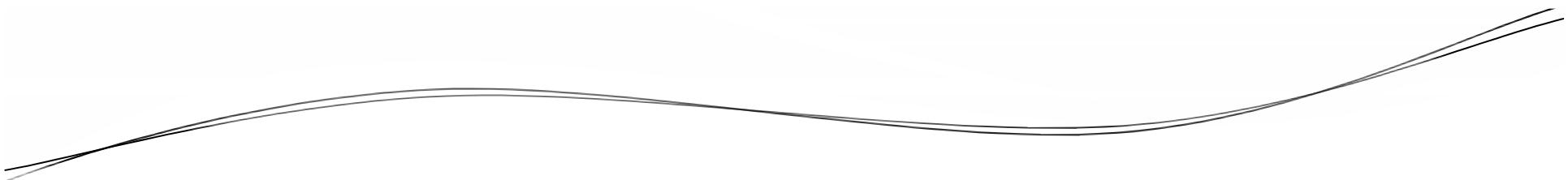
What is Spring?

- Spring is a lightweight dependency injection, aspect-oriented container and framework.



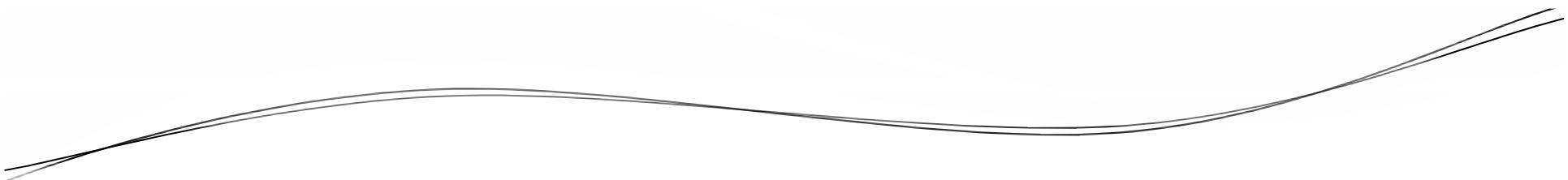
What is Spring?

- Spring is a Container as it creates the Java Components for your application and manages their life-cycle.

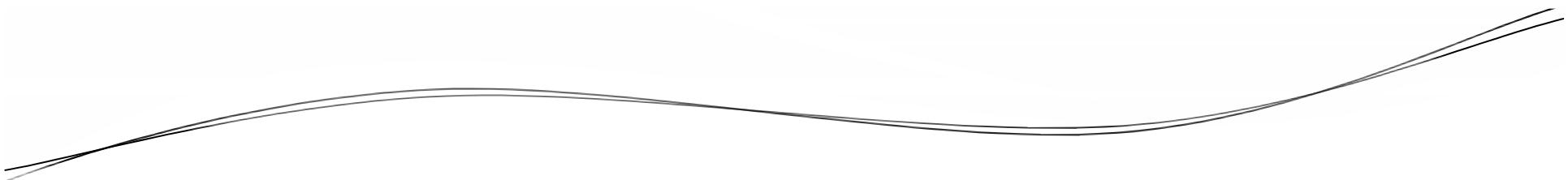


What is Spring Framework?

- It is lightweight in terms of size and overhead.
- It is distributed in a single ZIP file.

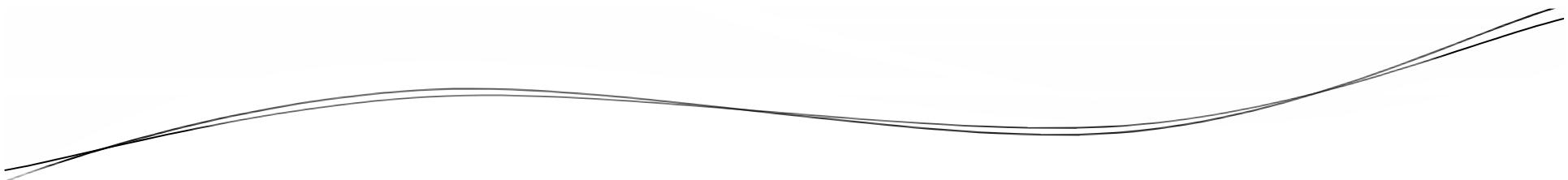


Why Spring Framework?



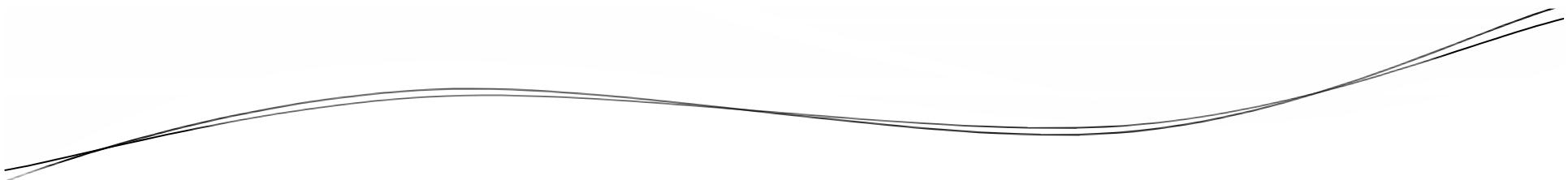
Why Spring Framework?

- Java components running inside Spring container have NO dependency on Spring specific Classes and Interfaces.

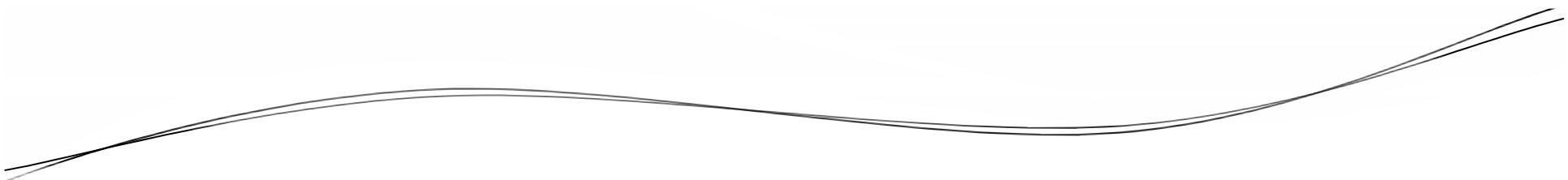


Why Spring Framework?

- Spring enables to build applications from “plain old Java objects” (POJOs) and to apply enterprise services to POJOs

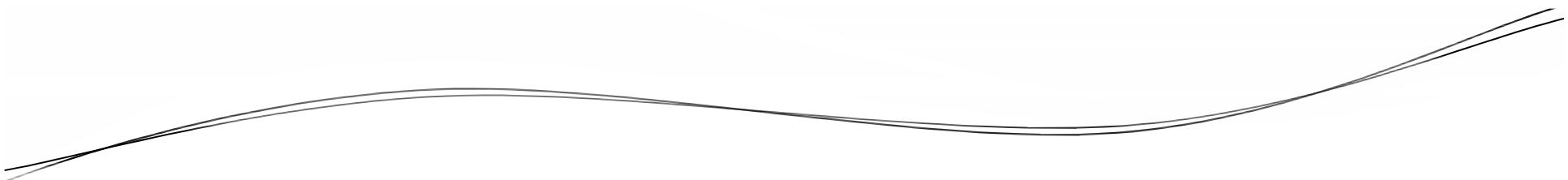


Spring Features



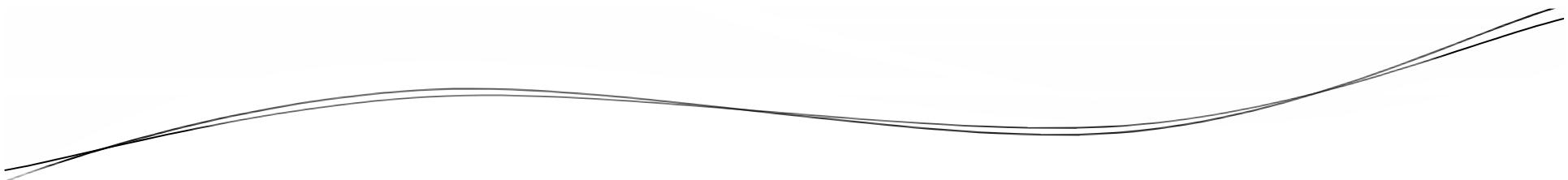
Spring Features

- Loose Coupling
- Dependency Injection
- Aspect Oriented Programming
- Data Access Support
- Support for Enterprise Services
- Integration Support
- MVC Support



Spring Features

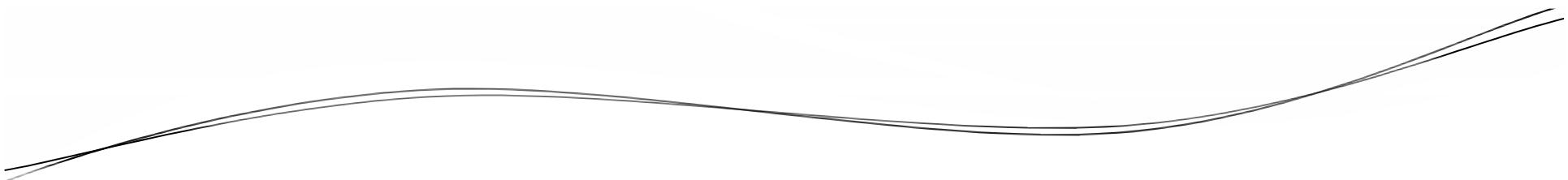
- Loose Coupling
 - Applications built using Spring are loosely coupled as it is not required to modify the Java source code in order to adopt changes in the configuration.



Spring Features

- Dependency Injection

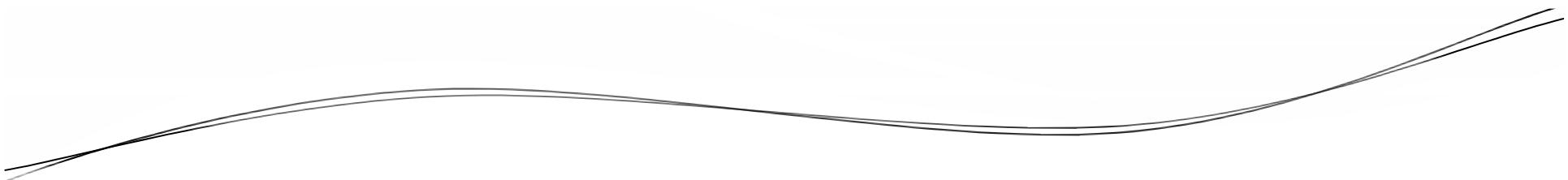
- A technique through which, the dependent objects are given to the components so that components need not create dependent objects themselves.



Spring Features

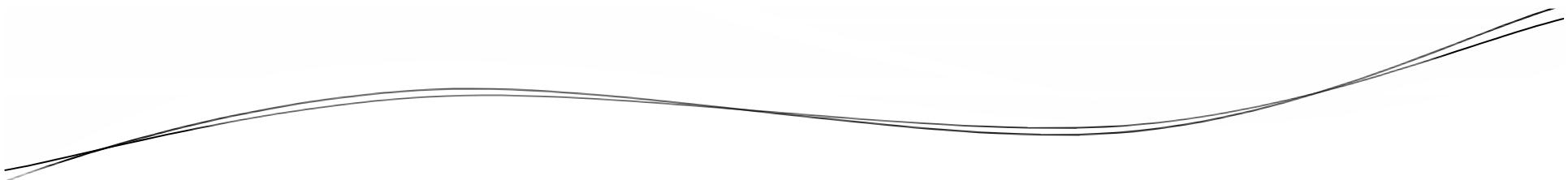
- Dependency Injection

- ```
public class Employee {
 private Address addr;
 //Some Code
}
```



# Spring Features

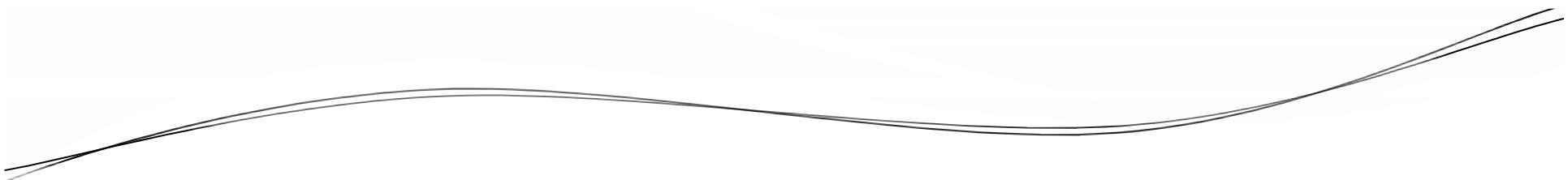
- At the time of creating Employee, the container injects the inner object (Address) inside the main component (Employee).



# Spring Features

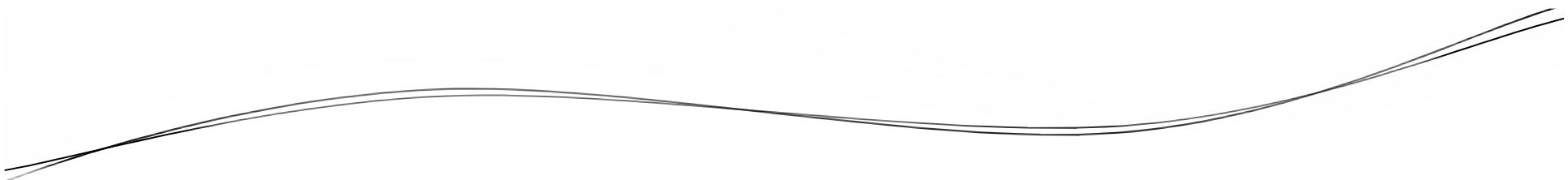
- Dependency Injection

- Dependency Injection (DI) is also known as Inversion Of Control (IoC) as now the control is with the Container to create the Dependent Object and inject it inside the Component Object.



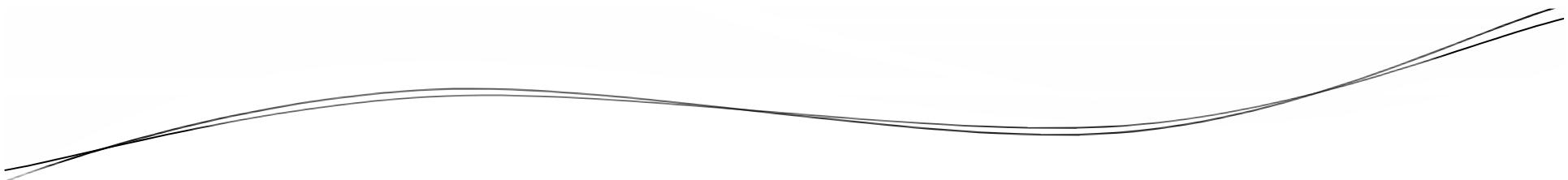
# Spring Features

- Aspect Oriented Programming
  - AOP is a programming model that promotes separation of business logic from the system concerns such as logging, transaction management, security, persistence etc.



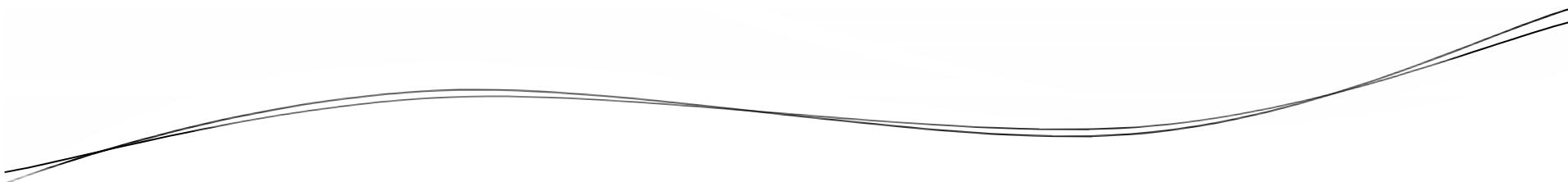
# Spring Features

- Aspect Oriented Programming
  - The code that implements these system wide concerns is NOT duplicated across multiple components.
  - The component development is simple as they can focus now only upon core business logic.



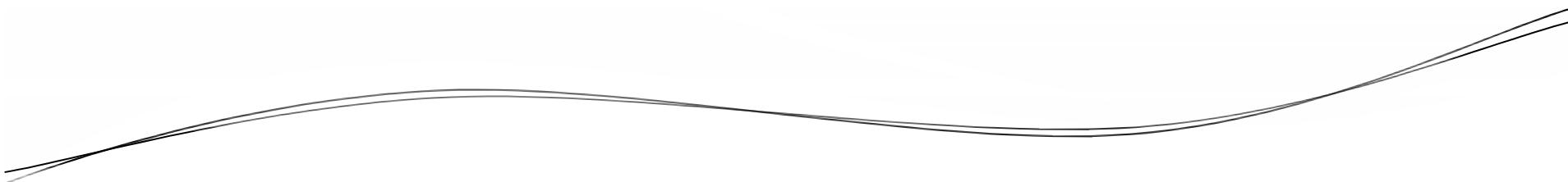
# Spring Features

- Data Access Support
  - Spring abstracts away the common code like opening and closing connections, so that the database code can be clean and simple.



# Spring Features

- Data Access Support
  - Spring doesn't attempt to implement its own ORM solution, but provides hooks into several popular ORM frameworks like Hibernate, JPA, iBATIS etc.

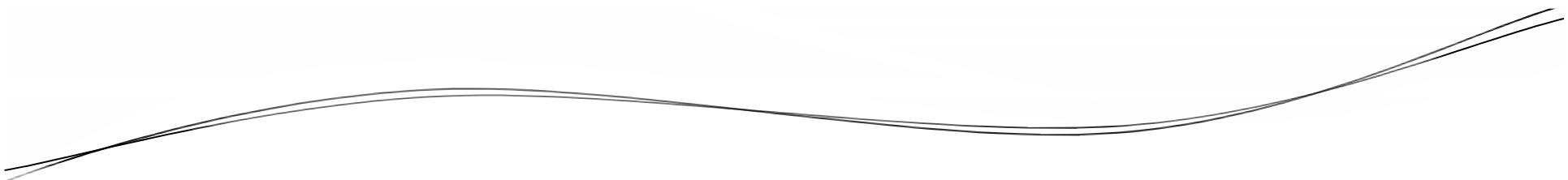


# Spring Features

- Support for Enterprise Services
  - An application built using Spring can acquire various enterprise level services like:
    - Transaction Management
    - Persistence
    - Asynchronous Messaging
    - Security
    - Task Scheduling

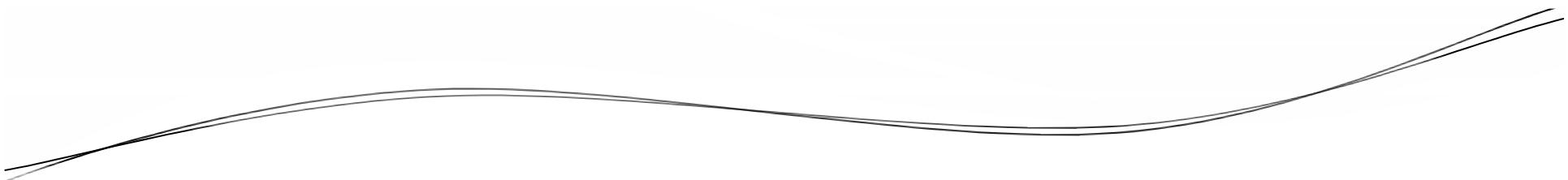
# Spring Features

- Integration Support
  - Spring provides support for integration with other technologies like EJB, JNDI, Web Services etc.
  - Already implemented services can be consumed very efficiently using Spring.



# Spring Features

- MVC Support
  - Spring comes with its own MVC framework and even it can integrated with existing popular MVC frameworks like Struts, JSF, Tapestry etc.

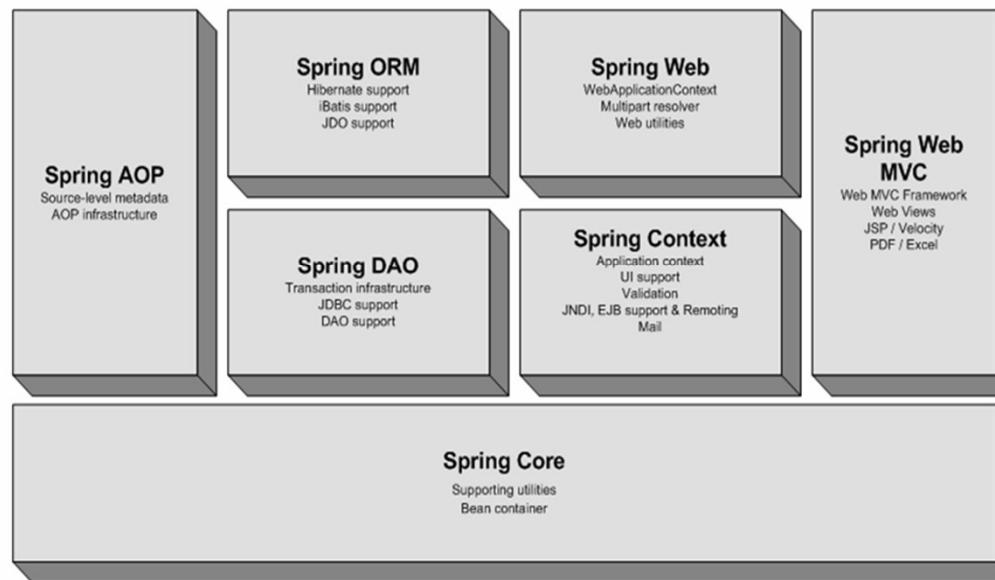


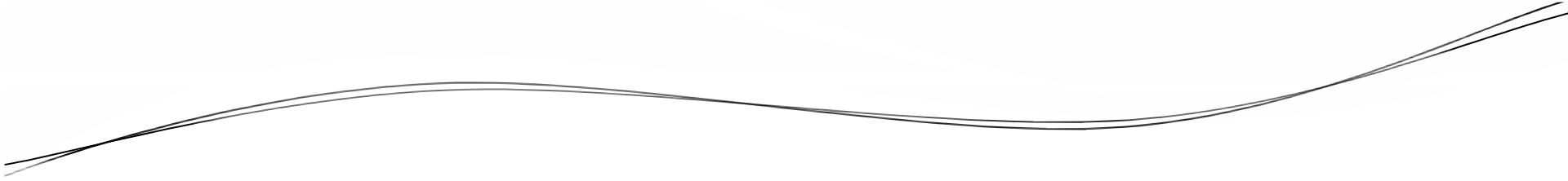
# ApplicationContext

- Built on the top of core container.
- Provides support for internationalization, application life cycle events and validation.

# Spring Framework – Core Modules

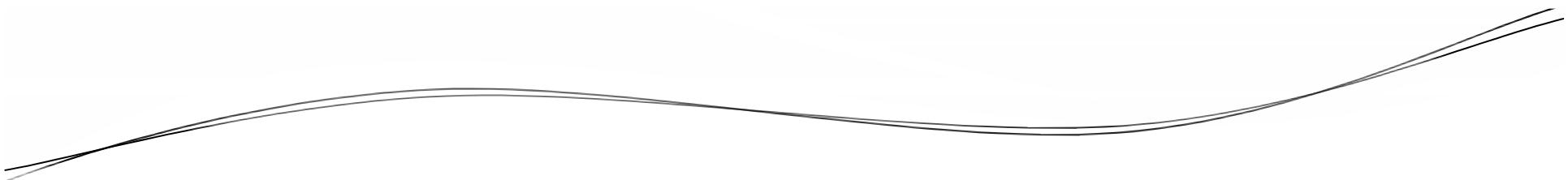
## Spring Architecture



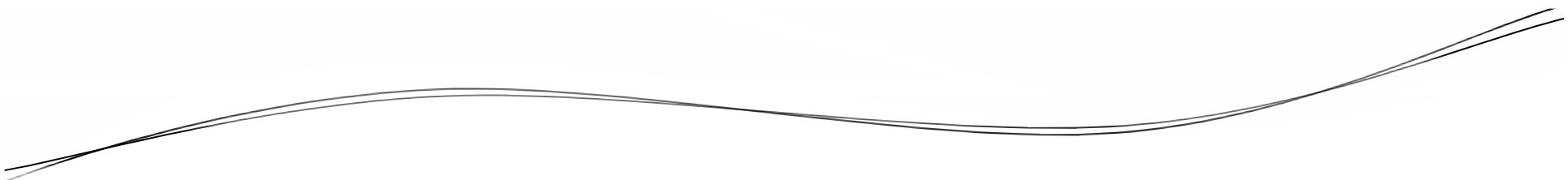


# Let's Summarize

- What is Spring
- Why Spring
- Spring Features
- Spring Modules
- Spring Architecture
- First Example

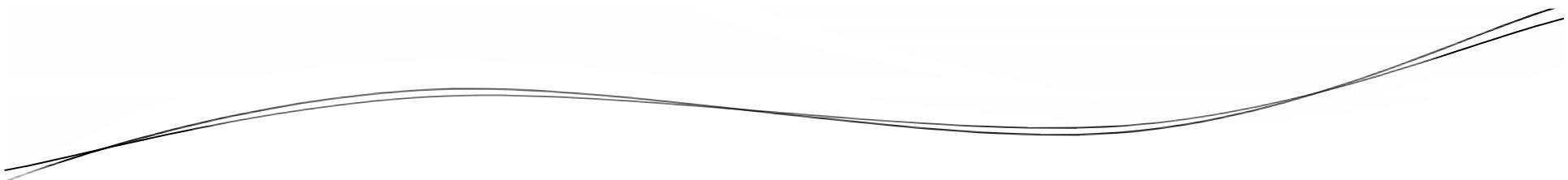


# **Annotation Based Configuration**

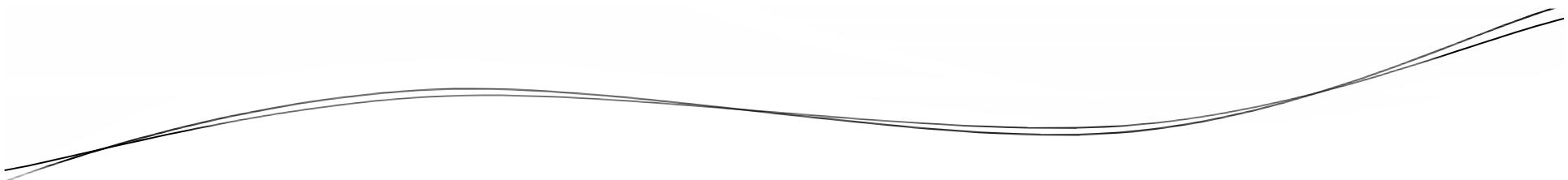


# Objectives

- Introduction to Spring's Annotation Support
- Bean Configuration
- Using Various Annotations

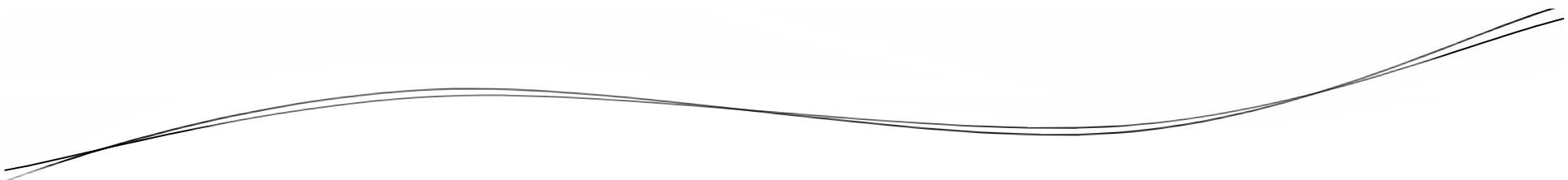


# Spring Annotations



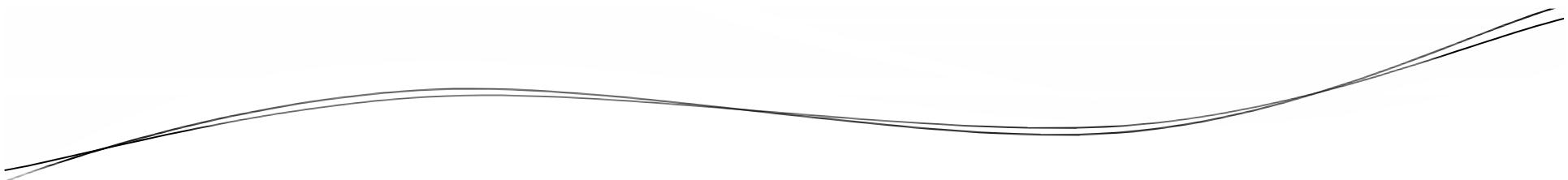
# **Spring Annotations**

- Spring Framework provides support for Annotation Based Metadata to handle RAD.
- Developers may discard XML totally and take full advantage of Spring Annotations.



# Spring Annotations

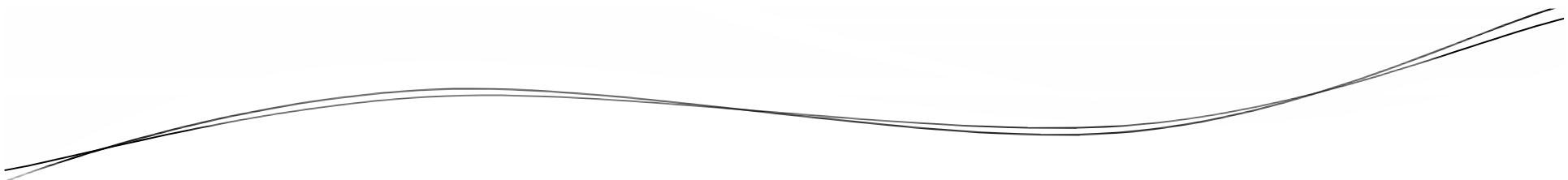
- In Annotation based configuration, there are further 2 options:
  - Java Based Configuration
  - Pure Annotation Based Configuration



# Configuring Beans

# Configuring Beans

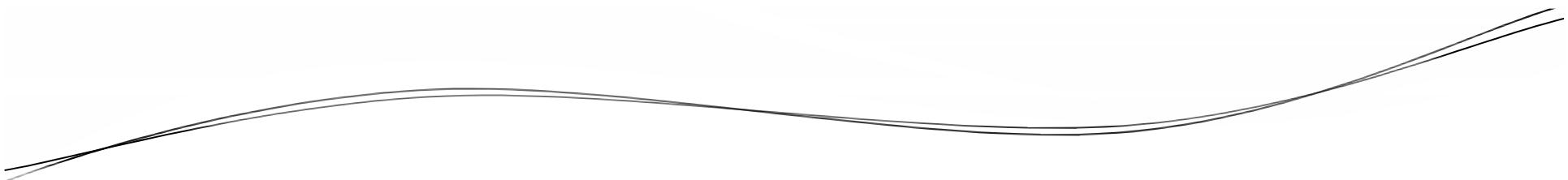
- To configure beans, Spring provides 2 basic annotations:
  - `@Configuration`
  - `@Bean`



# @Configuration

# **@Configuration**

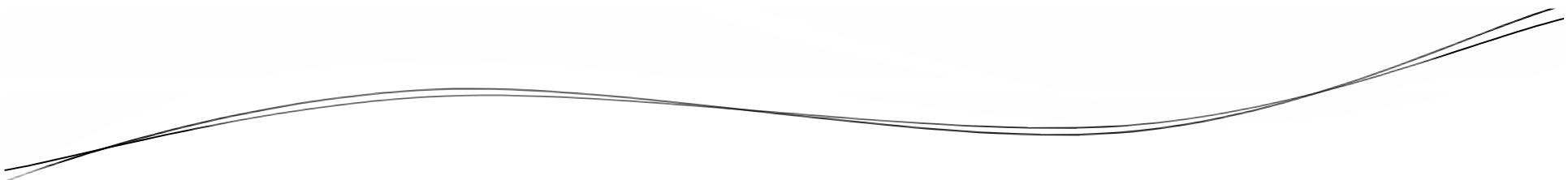
- Applied at the class level to introduce a class as a Configuration Unit.
- Classes annotated with @Configuration act as entry points of the spring configuration unit.



**@Bean**

## **@Bean**

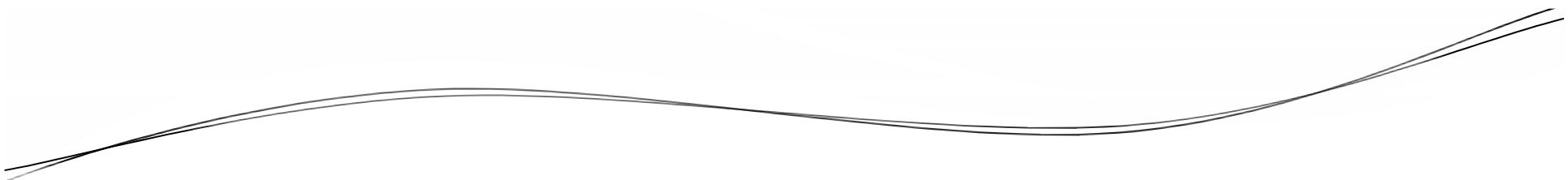
- Applied at the method level to indicate that a method is a Bean Creation Method.
- Objects returned by methods annotated with @Bean are treated as managed components in the Spring Environment.



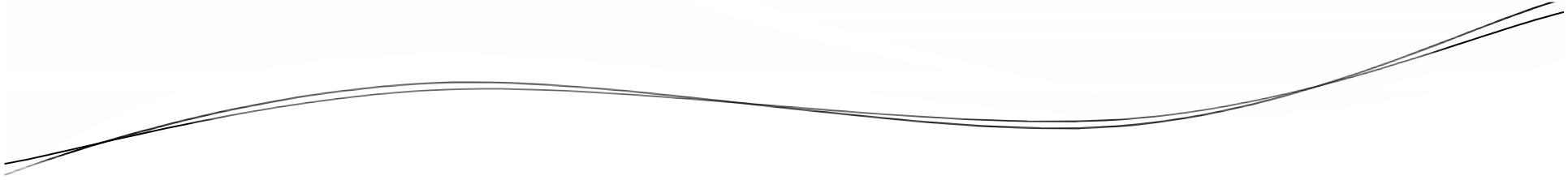
# **Retrieving Beans**

# Retrieving Beans

- Beans registered in the annotation based configuration unit are obtained using a class `AnnotationConfigApplicationContext`.

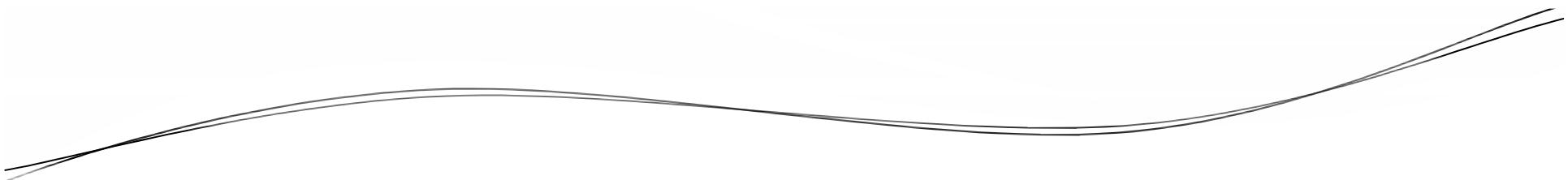


# **AnnotationConfigApplicationContext**



## **AnnotationConfigApplicationContext**

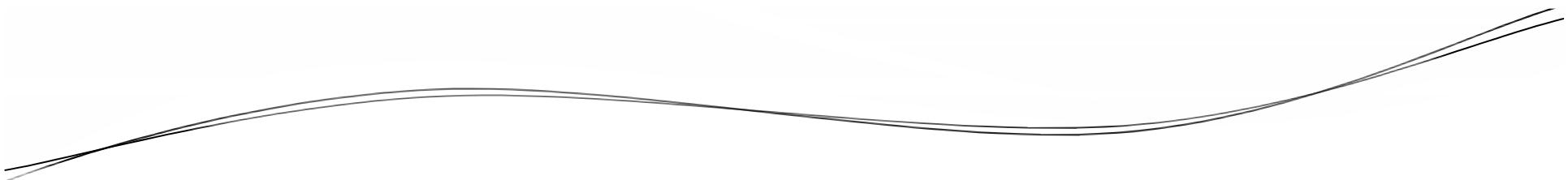
- A class used to register the configuration specific class so that beans can be obtained against their identities.



# Components

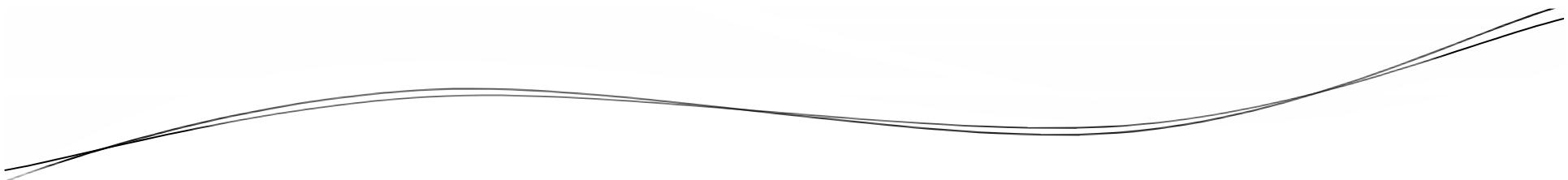
# Components

- @Bean annotation can be used to configure beans in the configuration unit.
- However, developer needs to create these objects explicitly.



# Components

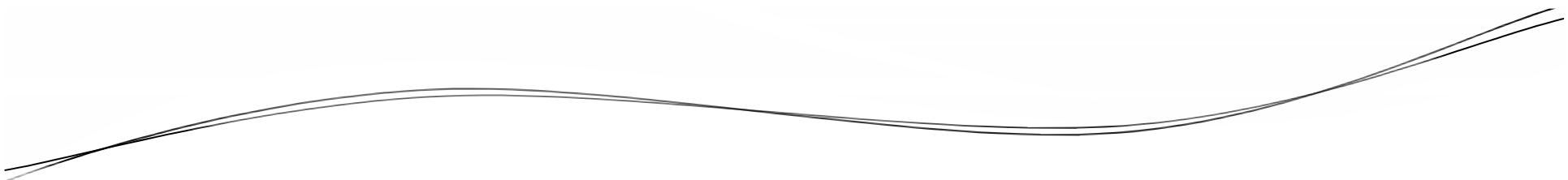
- To enable Spring to create Java Objects using Reflection API, Spring provides a stereotype annotation `@Component`.



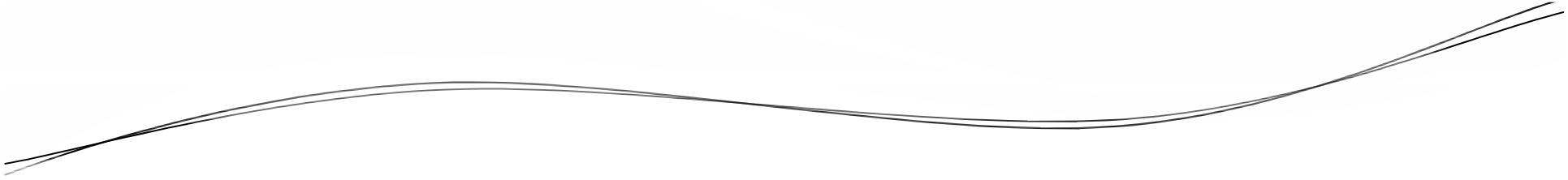
# Components

- @Component

Applied at the class level to mark that class as a Component class.

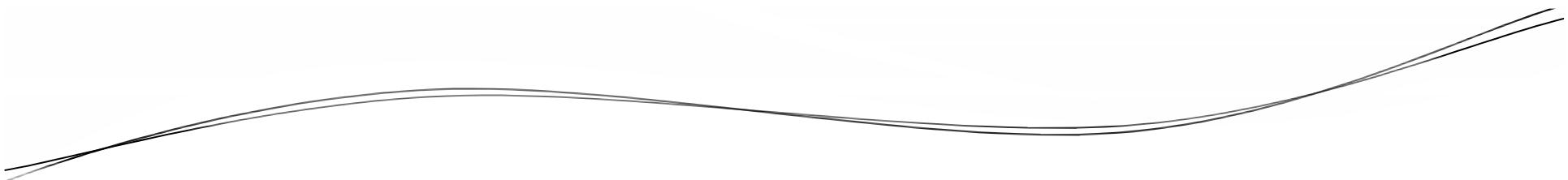


# Scanning Components



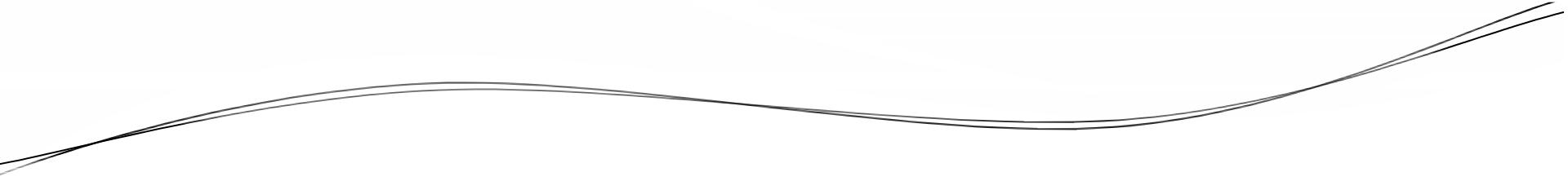
# Scanning Components

- Once a component is declared, it is to be scanned in the configuration unit and that is accomplished by an annotation `@ComponentScan`.

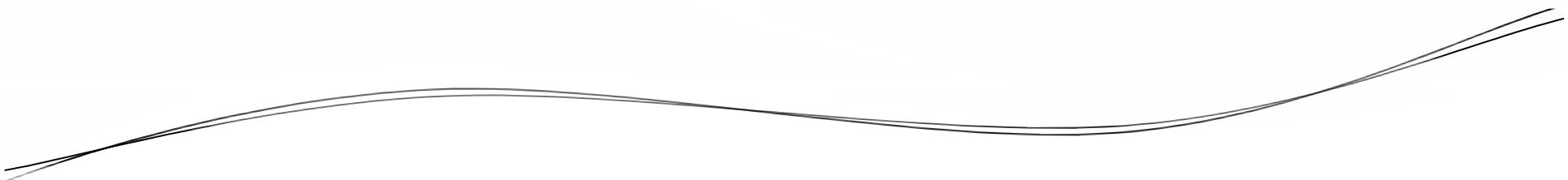


# Let's Summarize

- Spring's Annotation Support
- Bean Configuration
- Component Scanning
- Using Various Annotations

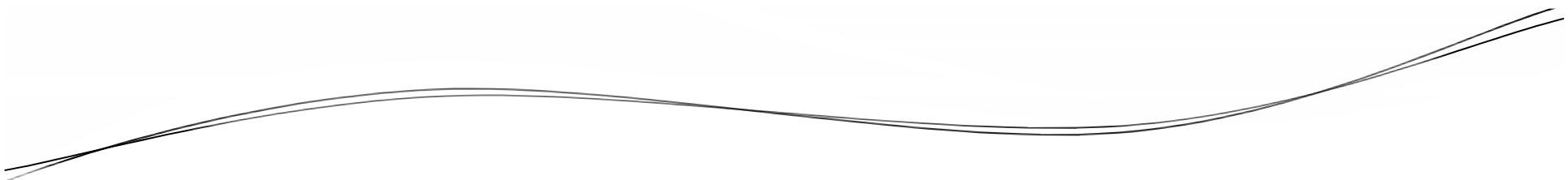


# AOP

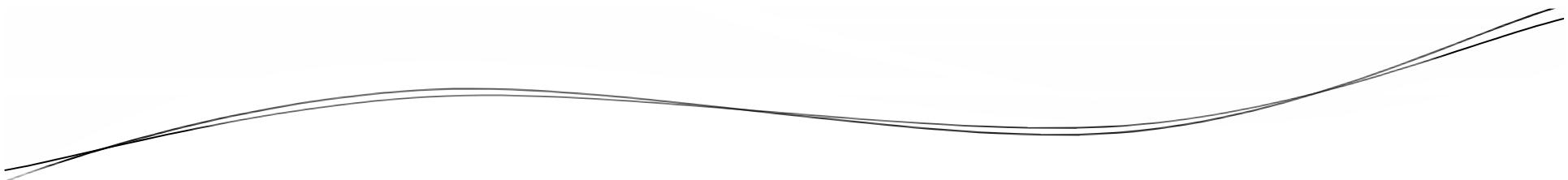


# Objectives

- Understand Aspect Oriented Programming.
- Why AOP
- AOP Terminologies
- Weaving
- Implementing AOP

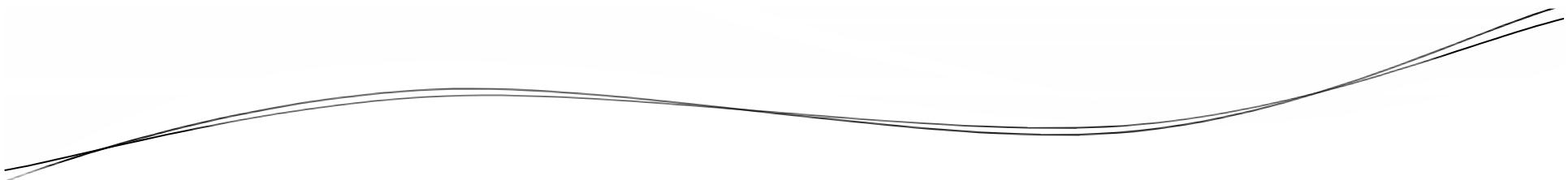


# What is AOP



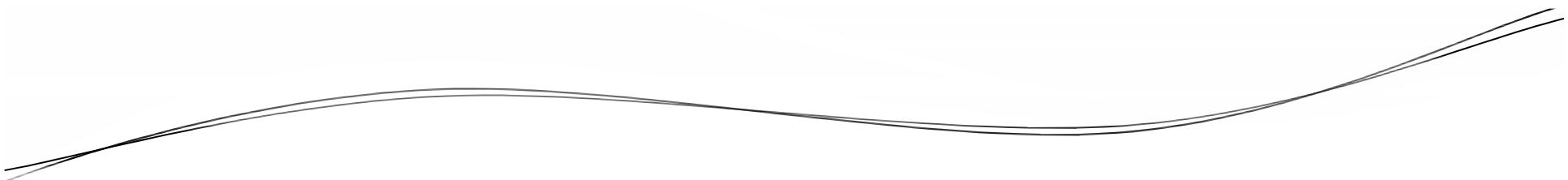
# What is AOP

- AOP is a programming model that promotes separation of business logic from cross-cutting concerns.



# What is AOP

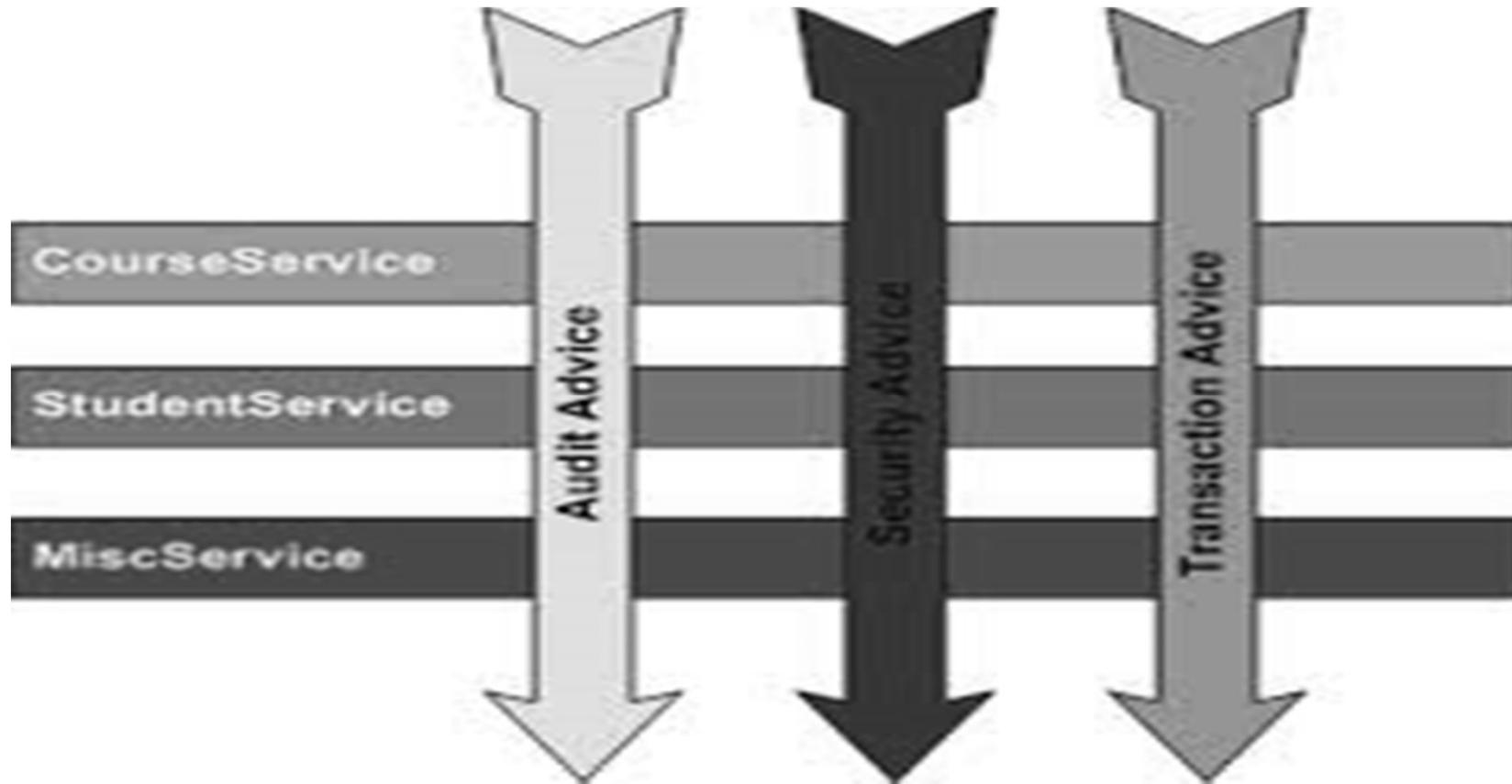
- A cross-cutting concern can be described as any functionality that affects multiple points of an application.

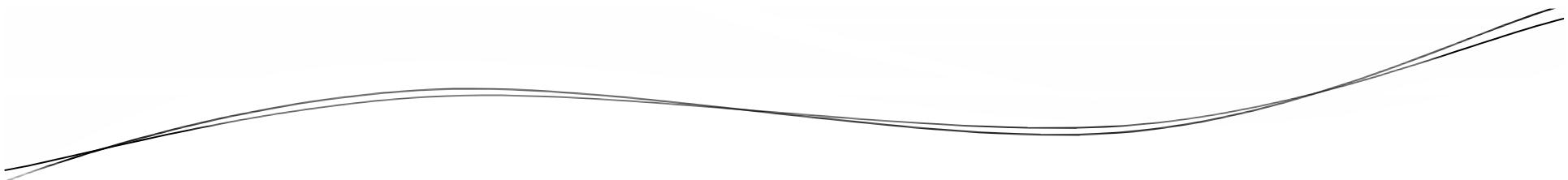


# What is AOP

- These cross-cutting concerns can be modularized into special objects called Aspects.

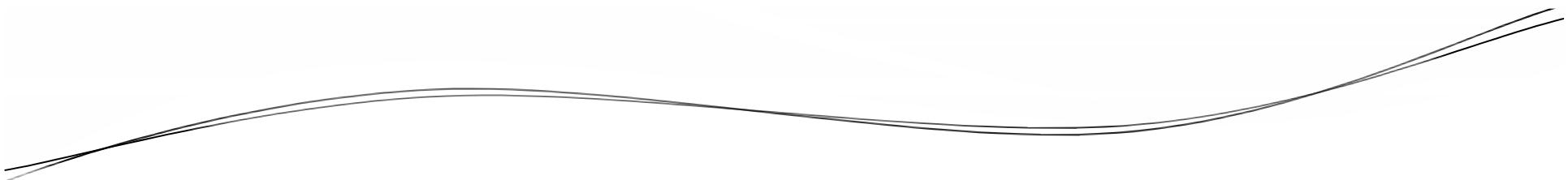
# AOP – Big Picture



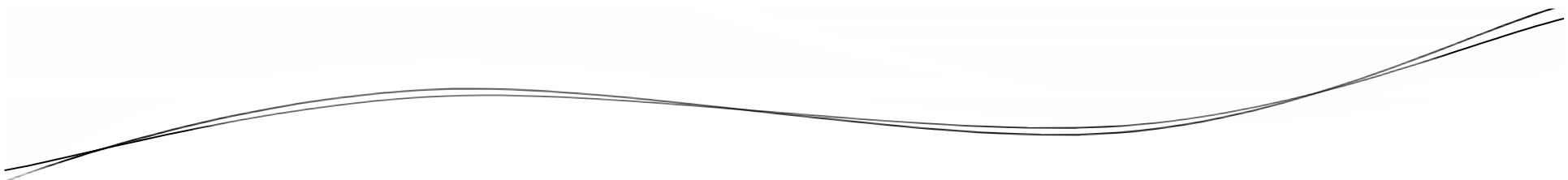


# Benifits

- Aspects offer an alternative to inheritance and delegation that can be cleaner in many circumstances.
- Secondary concerns are decoupled from primary concerns.
- Allows developers to focus upon business logic.



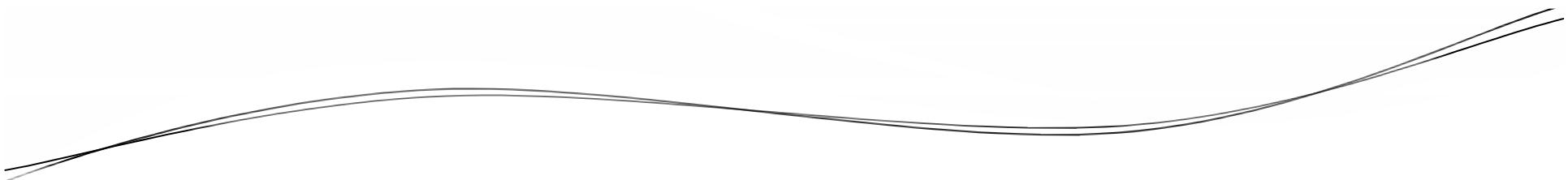
# AOP Terminology



# AOP Terminology

- There are 5 terms involved in Spring AOP:

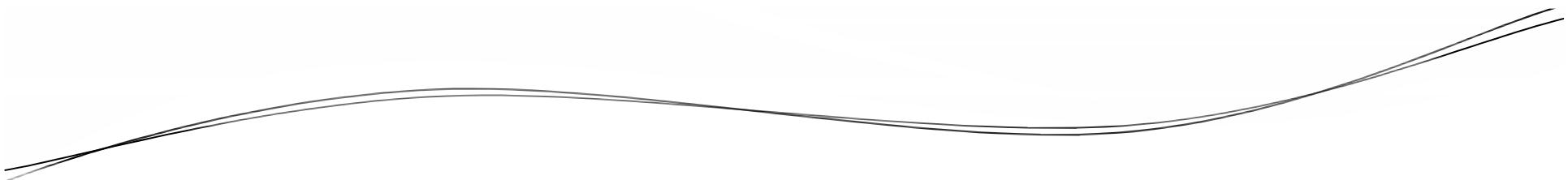
- Advice
- Joinpoint
- Pointcut
- Aspect
- Weaving



# AOP Terminology

- Advice

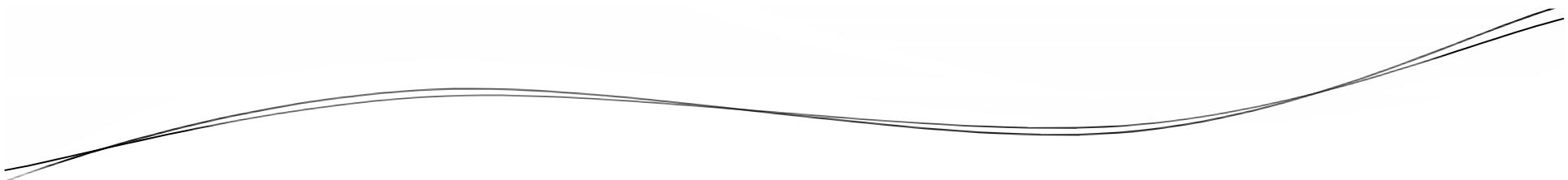
- Aspects have a purpose, a job they are meant to do.
- The job of an aspect is called advice.
- It defines both: WHAT and WHEN of an aspect.



# AOP Terminology

- JoinPoint

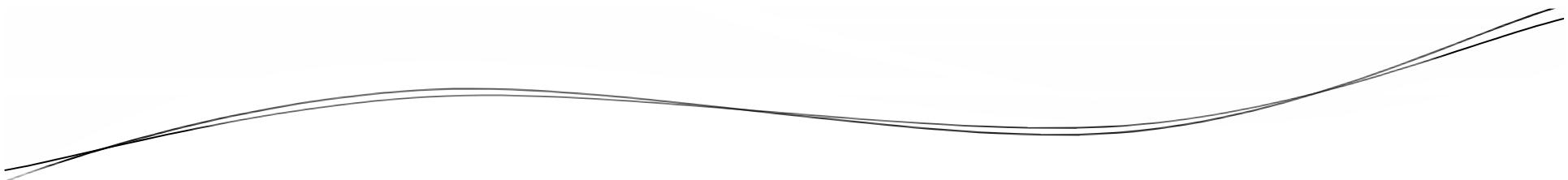
- A joinpoint is a point in the execution of the application where an aspect can be plugged in.



# AOP Terminology

- Pointcut

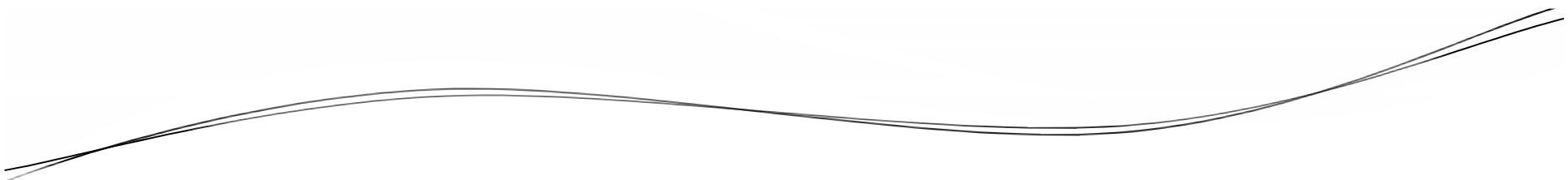
- If advice defines WHAT and WHEN of an aspect, then pointcut defines WHERE.
- A pointcut definition matches one or more joinpoints at which advice should be woven.



# AOP Terminology

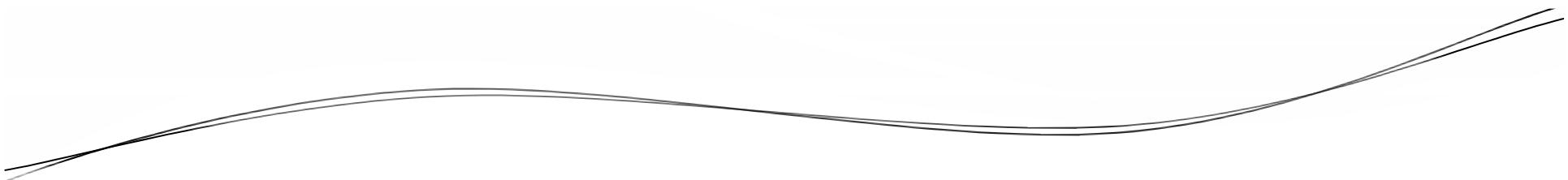
- Aspect

- An aspect is the merger of advice and pointcut.
- It specifies what to do, when to do and where to do.



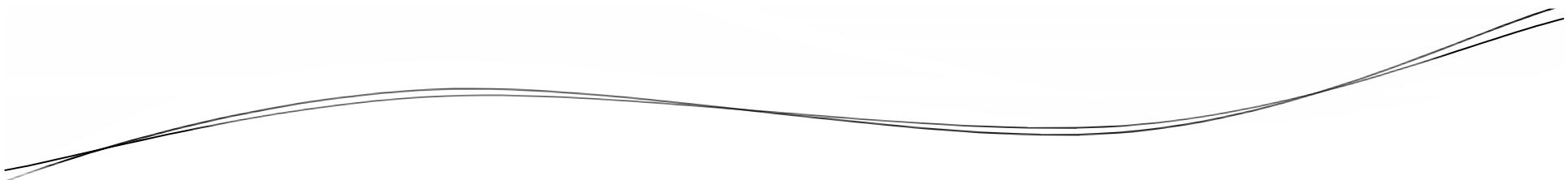
# Weaving

- Weaving is the process of applying aspects to a target object to create a new, proxied object.
- The aspects are woven into the target object at the specified join points.



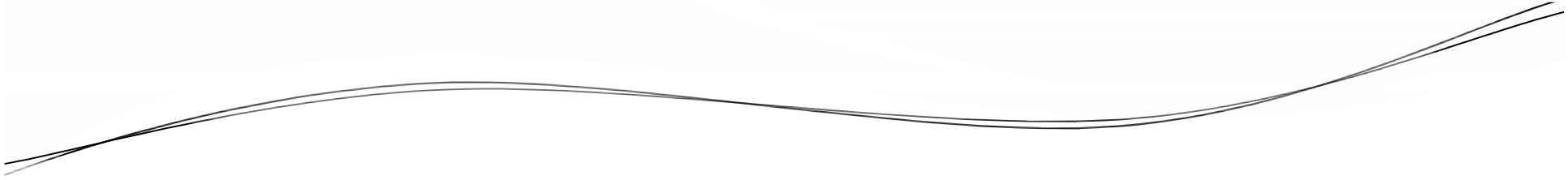
# Weaving

- Weaving can take place at three different points:
  - Compile time
  - Classload time
  - Runtime



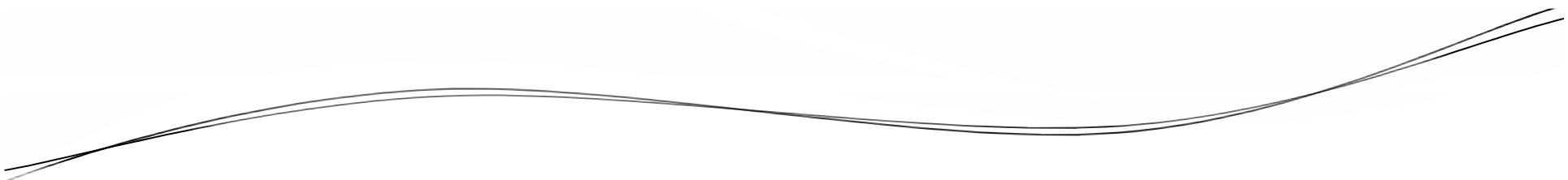
# Weaving

- Compile time
  - Aspects are woven in when the target class is compiled.



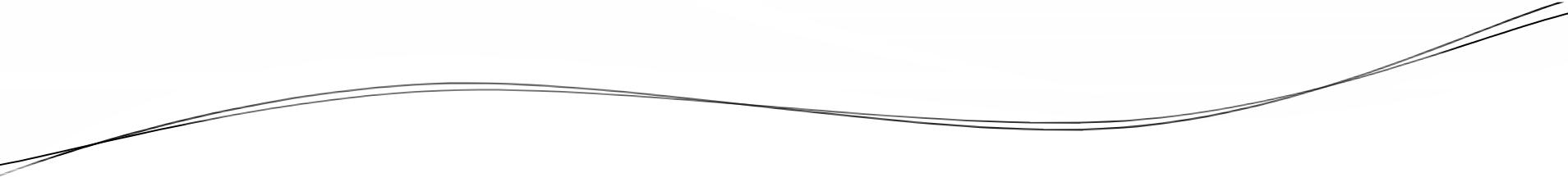
# Weaving

- Classload Time
  - Aspects are woven in when the target class is loaded into the JVM.
  - This requires a special class loader that enhances the byte code before the class is introduced to the application.

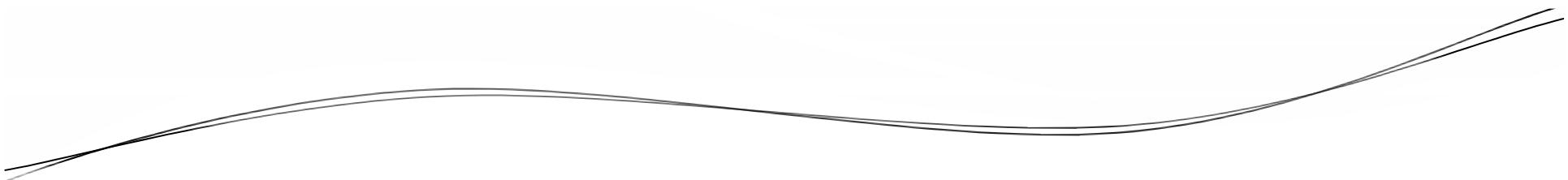


# Weaving

- Runtime
  - Aspects are woven in sometime during the execution of the application.
  - Typically, an AOP container will dynamically generate a proxy object that will delegate to the target object while weaving in the aspects.
  - This is how Spring AOP aspects are woven.

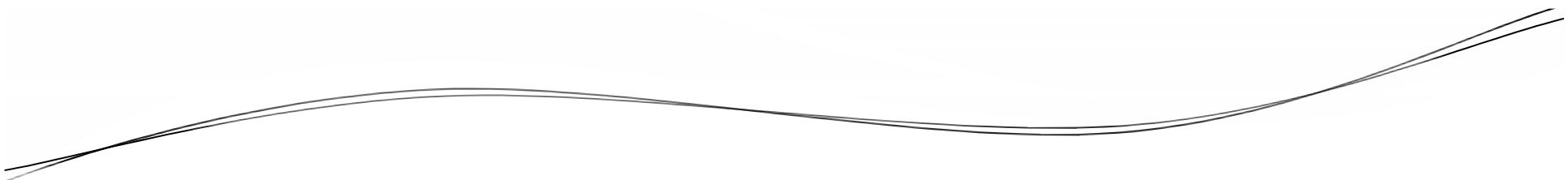


# **Spring AOP support**



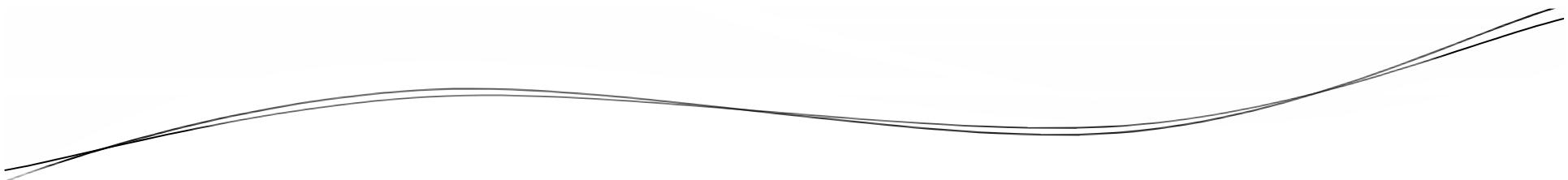
# Spring AOP support

- Spring Advices are written in Java.
- Spring applies an Advice to an Object at runtime, by wrapping them with a proxy class.

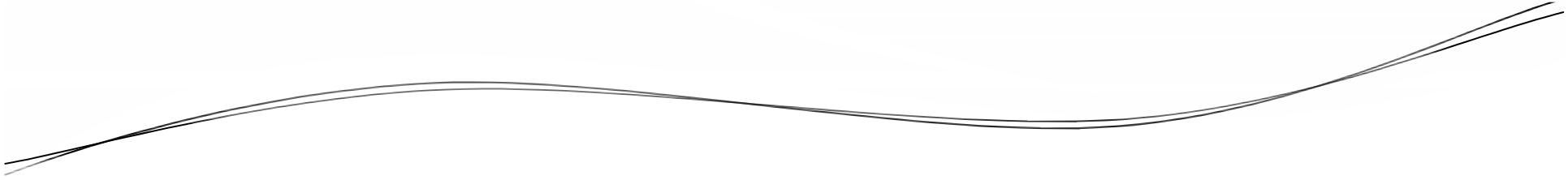


# Spring AOP support

- Between the time that the proxy intercepts the method call and the time it invokes the target bean's method, the proxy performs the aspect logic.
- Spring only supports method joinpoints.

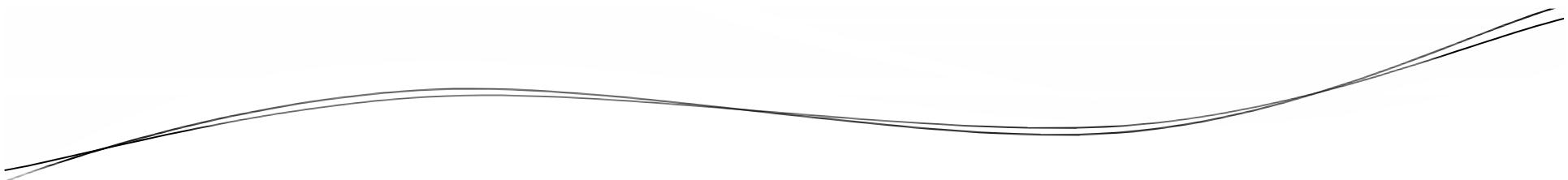


# **Types of Spring AOP Advices**



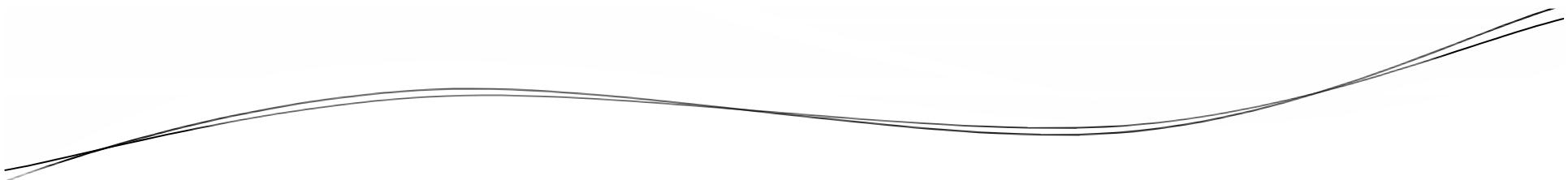
# Types of Spring AOP Advices

- Before Advice
- After returning Advice
- After throwing Advice
- After Advice
- Around Advice



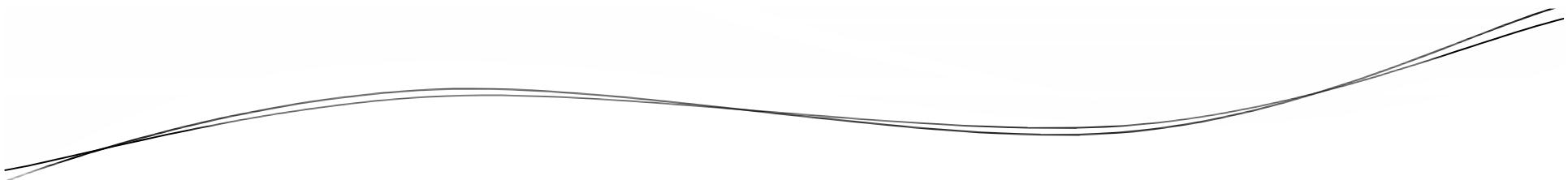
# Before Advice

- Causes a method to be invoked before the invocation of a target method.



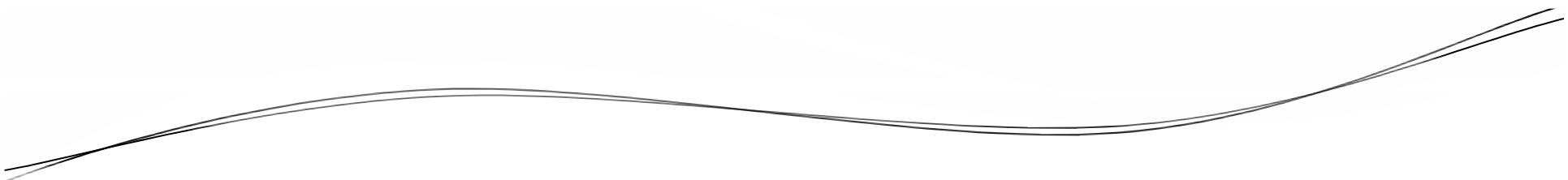
# **After Returning Advice**

- Causes a method to be invoked after successful invocation of a target method.



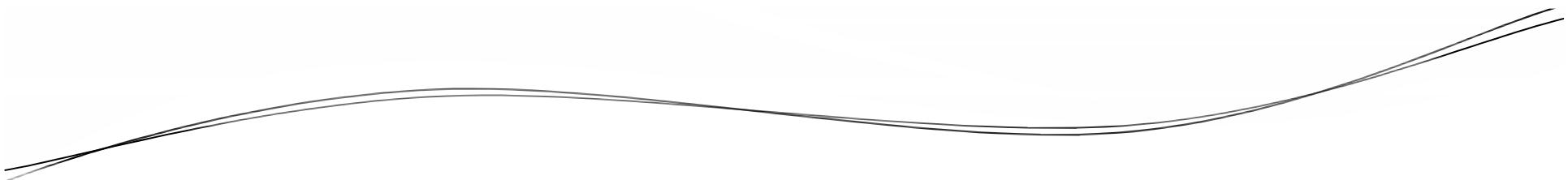
# After Throwing Advice

- Causes a method to be invoked if an exception is occurred during an execution of the target method.



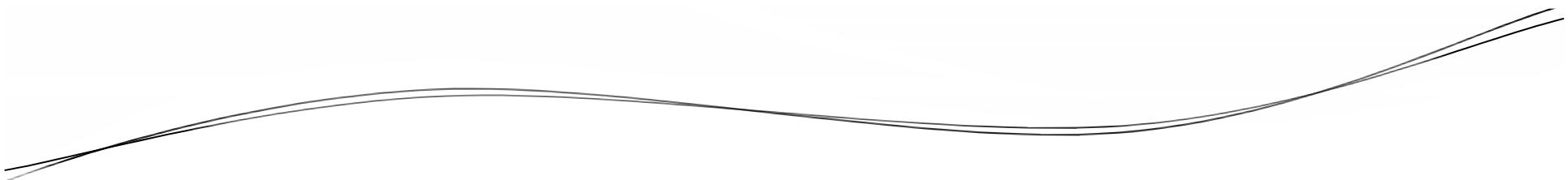
# After Advice

- Causes a method to be invoked irrespective of whether a target method completes successfully or not.

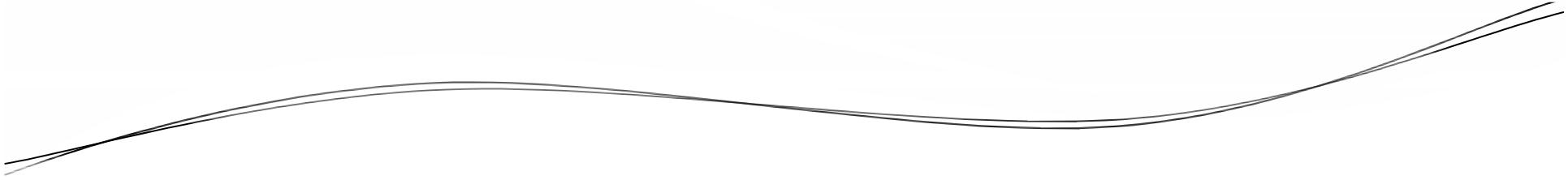


# Around Advice

- A single advice that wraps up all types of advices.
- Provides a single method through which other advice methods are invoked.

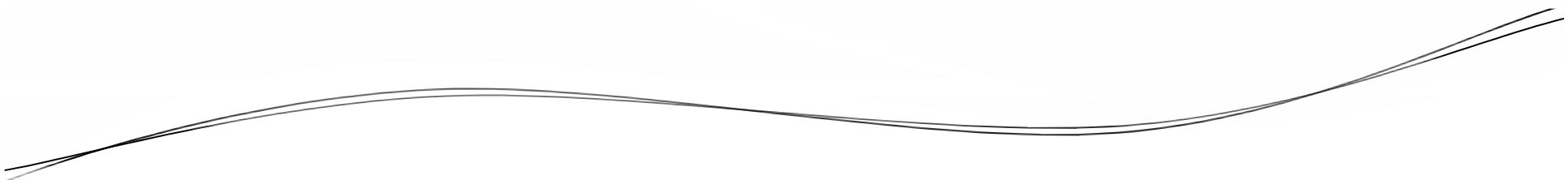


# Enabling Proxying



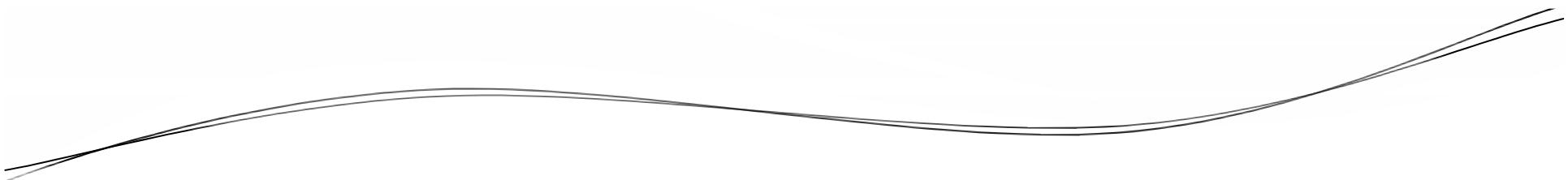
# Enabling Proxying

- Since in AOP, spring performs weaving with the help of proxies, it's necessary to enable Proxy in the application.

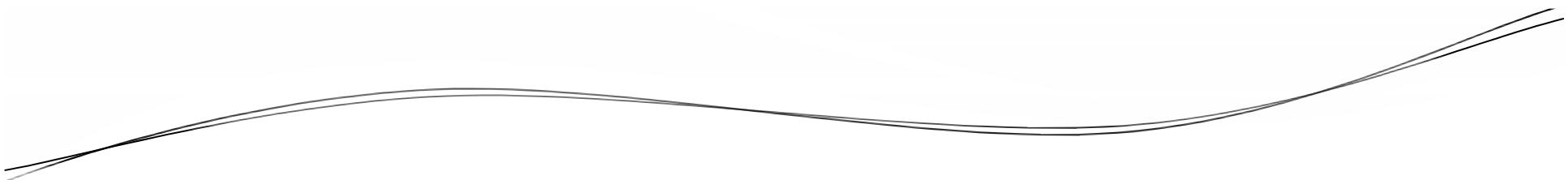


# Enabling Proxying

- To enable proxy, the configuration specific class must be annotated with `@EnableAspectJAutoProxy`.

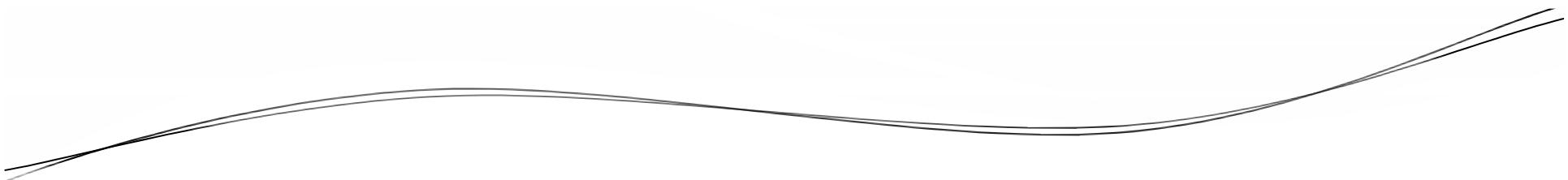


# **Spring's AOP Annotations**



# **Spring's AOP Annotations**

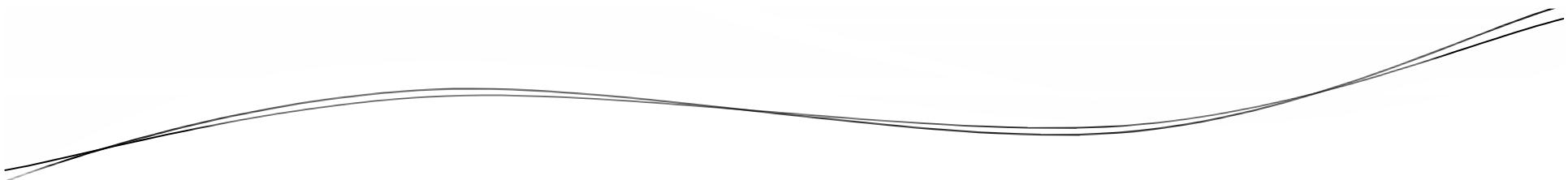
- Spring comes with Annotation Based Support, through AspectJ annotations for AOP.



# Spring's AOP Annotations

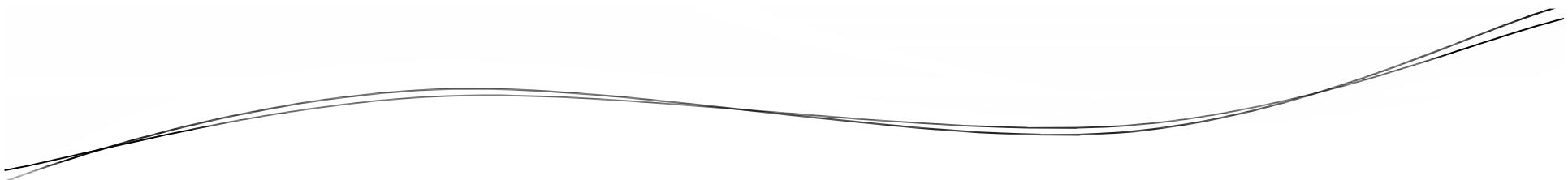
- AOP Annotations:

- @Aspect
- @Before
- @AfterReturning
- @AfterThrowing
- @After
- @Pointcut
- @Around



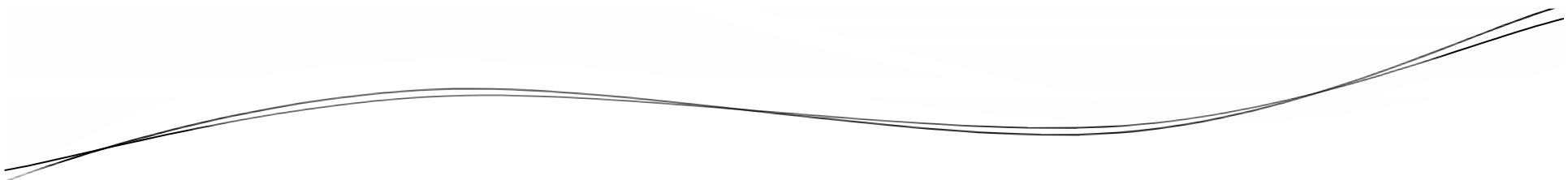
## **@Aspect**

- Applied at the class level.
- Tells Spring that the class is not just a POJO, rather it's an aspect.
- Classes annotated with @Aspect will only be considered as *Aspects*.



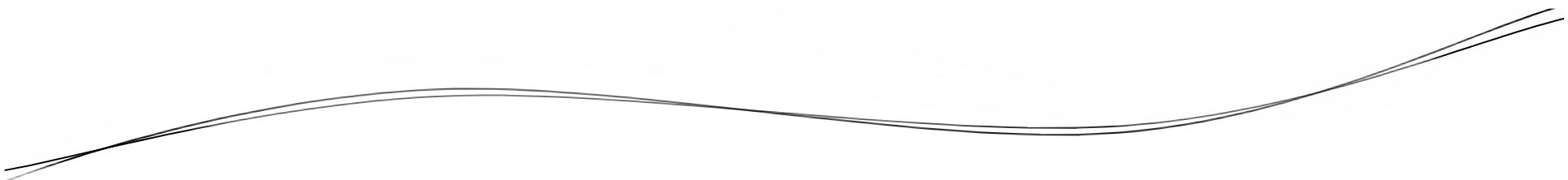
## **@Before**

- Applied at the method level.
- Annotated methods will be called before the execution of advised method.



## **@AfterReturning**

- Applied at the method level.
- Annotated methods will be called after the successful return of the advised method.

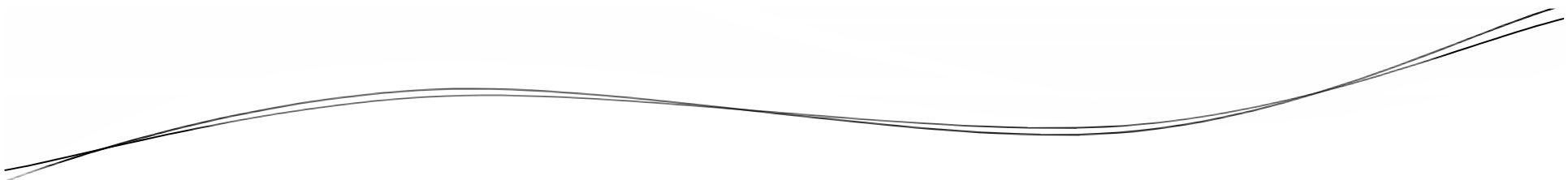


## **@AfterThrowing**

- Applied at the method level.
- Annotated methods will be called if some exception is raised during the execution of the advised method.

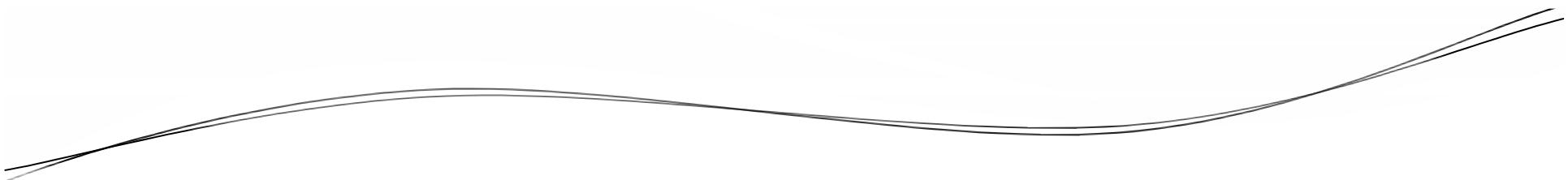
## **@After**

- Applied at the method level.
- Annotated methods will be called after the execution of the advised method, irrespective of whether the advised method returns successfully or not.



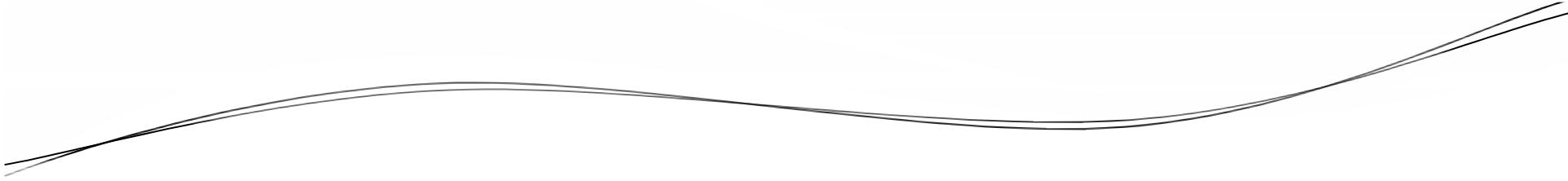
## **@Pointcut**

- Used to define a pointcut.
- Applied at the method level to mark the method as a pointcut.
- The marked method can be used for further referencing.



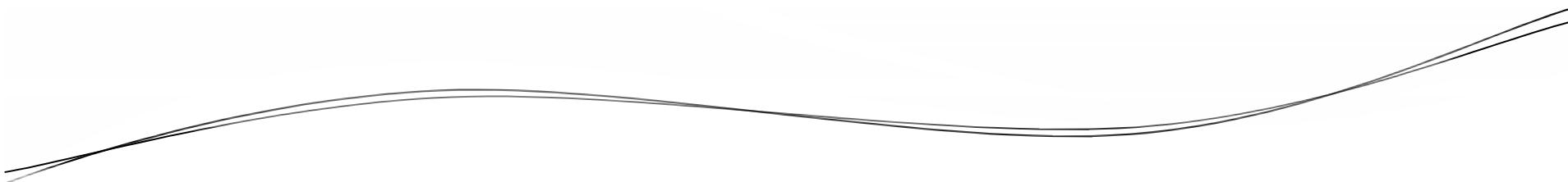
## **@Around**

- Applied at the method level to mark the method as a Around Advice.
- Uses a Pointcut expression to apply the aspect.

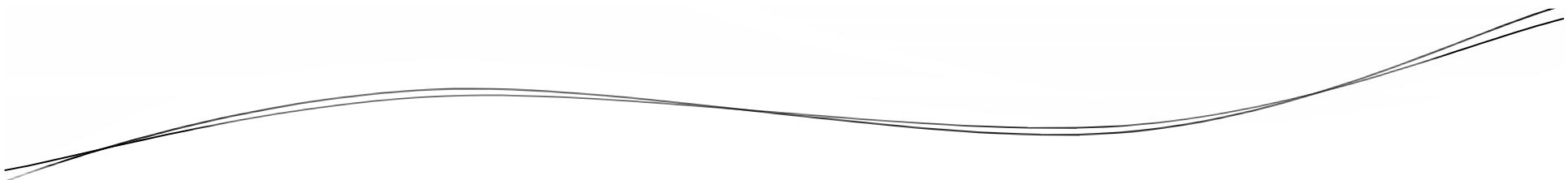


# Let's Summarize

- What is AOP
- Why AOP
- AOP Terminologies
- Types of Advices
- Weaving
- Implementing AOP

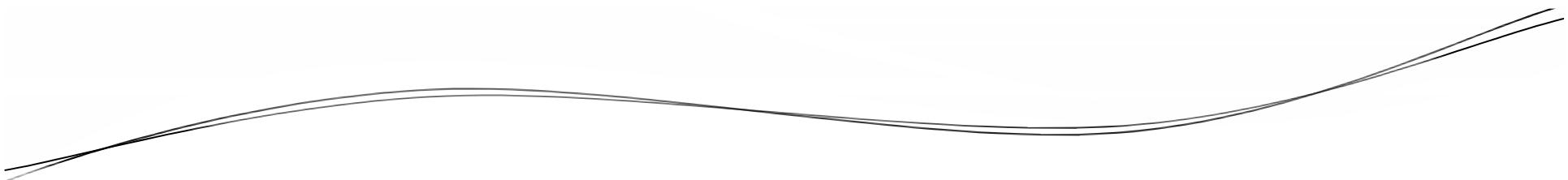


# **Bean Management**



# Objectives

- Working with ApplicationContext
- Bean Life Cycle
- Understanding Dependency Injections
- Bean Scopes
- Auto-wiring

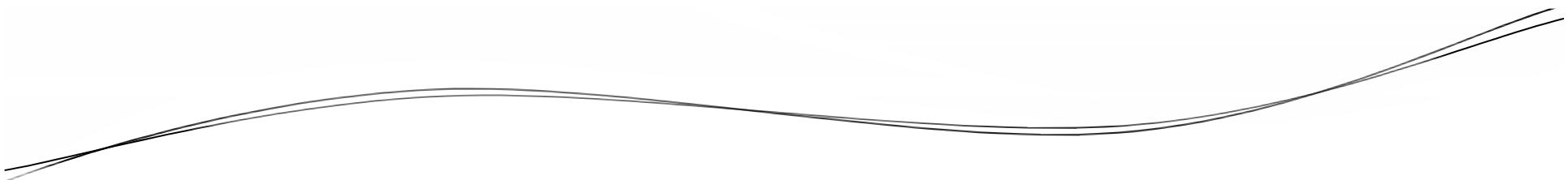


# ApplicationContext

- In a Spring-based application, the application objects live within the Spring Container.
  - The container creates the objects, wire them together, configure them, and manage their lifecycle.

# ApplicationContext

- The *ApplicationContext* interface standardizes the Spring bean container behavior.
  - Its implementations form the simple container, providing basic support for DI.
  - Supports I18N.
  - Supports Event Management.



# ApplicationContext

- Spring comes with several flavors of *ApplicationContext*.
  - *FileSystemXmlApplicationContext*
  - *ClassPathXmlApplicationContext*
  - *XmlWebApplicationContext*

# FileSystemXmlApplicationContext

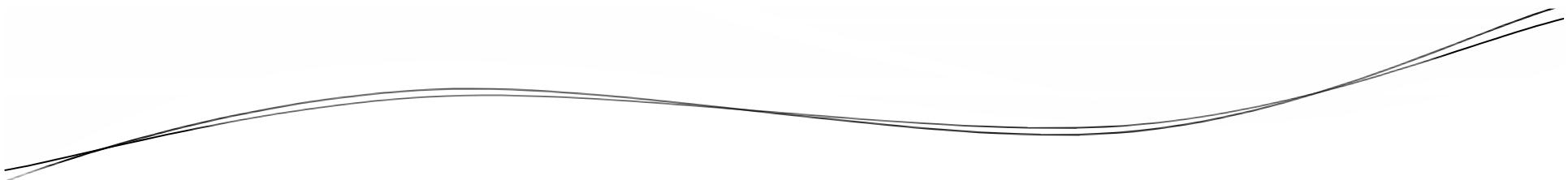
- Loads a context definition from an XML file located in the file system.
- E.g.

```
ApplicationContext ctx;
String file = "c:/config.xml";
ctx =
new
FileSystemXmlApplicationContext(file);
```

# ClassPathXmlApplicationContext

- Loads a context definition from an XML file located in the classpath.
- E.g.

```
ApplicationContext ctx;
String file = "config.xml";
ctx =
new ClassPathXmlApplicationContext(file);
```



# **XmlWebApplicationContext**

- Loads a context definition from an XML file contained within a web application.
- Used in Spring MVC environment.

# Bean Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
 <bean id="emp"
 class="com.emp.Employee">
 </bean>
</beans>
```

# Bean Configuration File

- **bean:** The Element which is the most basic configuration unit in Spring. It tells Spring Container to create an Object.
- **id:** The Attribute which gives the bean a unique name by which it can be accessed.
- **class:** The Attribute which tells Spring the type of a Bean.

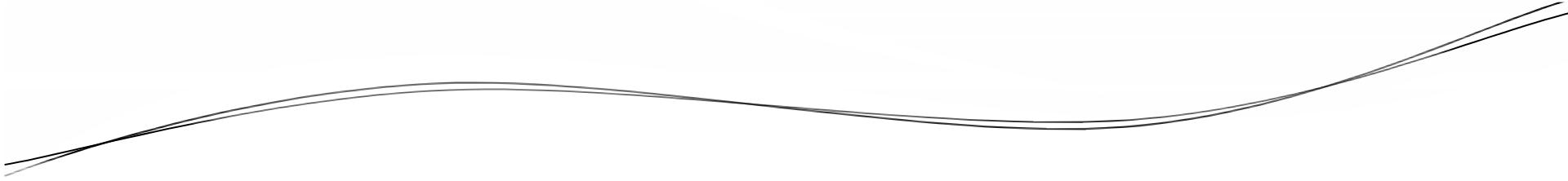
# Accessing Bean

```
ApplicationContext ctx;
String file = "c:/config.xml";
ctx = new
FileSystemXmlApplicationContext(file);
```

```
GreetingService service;
service =
(GreetingService) ctx.getBean("greet");
```

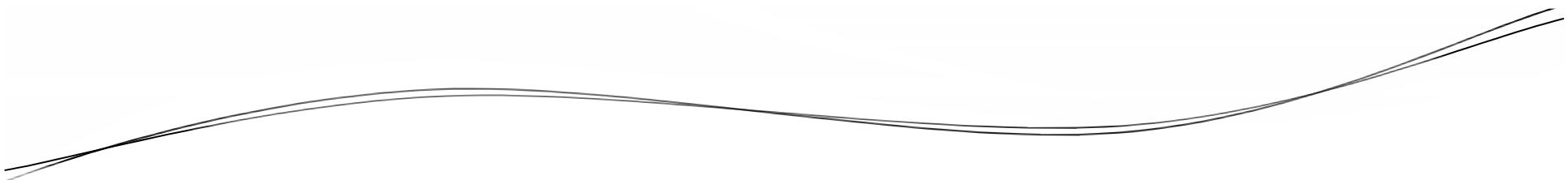
# How DI/IoC Container works

- In Inversion Of Control (IoC), control is inverted back to the Container to support the object dependencies.
- IoC container creates the POJO objects and provides dependencies to them.
  - These POJO objects are not tied to any framework.
- The declarative configuration for POJO objects is defined with unique identities (id) in XML.
  - These are known as bean definitions.



# How DI/IoC Container works

- The IoC container at runtime identifies POJO bean definitions, creates bean objects and returns them to the Application.
- IoC container manages dependencies of the objects.



# Injecting the Dependencies

- The IoC container will Inject the dependencies in three ways
  - Create the POJO objects by using no-argument constructor and injecting the dependent properties by calling the setter methods.

# Injecting the Dependencies

- The IoC container will Inject the dependencies in three ways
  - Create the POJO objects by using no-argument constructor and injecting the dependent properties by calling the setter methods.
  - Create the POJO objects by using parameterized constructors and injecting the dependent properties through the constructor.

# Injecting the Dependencies

- The IoC container will Inject the dependencies in three ways
  - Create the POJO objects by using no-argument constructor and injecting the dependent properties by calling the setter methods.
  - Create the POJO objects by using parameterized constructors and injecting the dependent properties through the constructor.
  - Implements the method Injection.

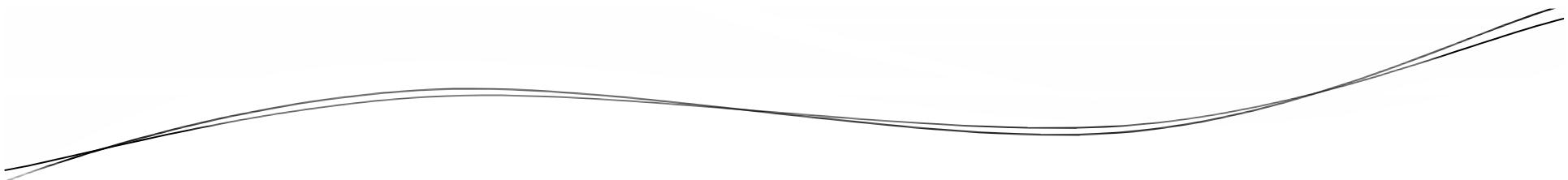
# Injecting Properties by calling Setters

```
public Class Employee {
 private String fname, lname;
 public Employee() {}

 public void setFname(String f) {
 fname = f;
 }
 public void setLname(String l) {
 lname = l;
 }
}
```

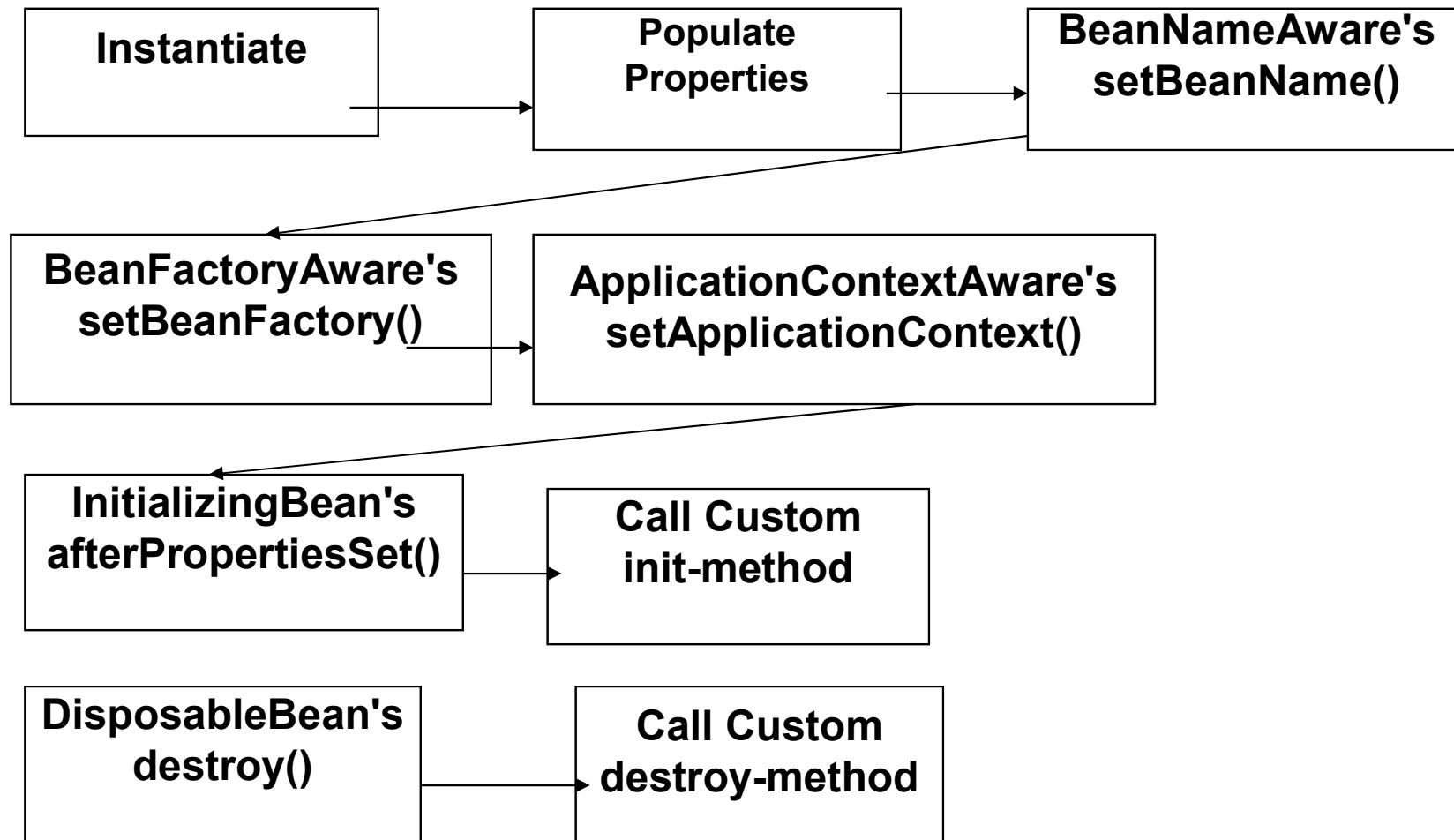
# Injecting Properties by calling Setters

```
<bean id="emp"
 class="com.emp.Employee">
 <property name="fname" value="a"/>
 <property name="lname" value="b"/>
</bean>
```



# **Bean LifeCycle**

# Bean LifeCycle



# Bean Life Cycle

- Instantiate: Spring instantiates the bean.
- Populate properties: Spring injects the bean's properties.
- Set bean name: If the bean implements BeanNameAware, Spring passes the bean's ID to setBeanName().

# Bean Life Cycle

- Set Application Context: If the bean implements `ApplicationContextAware`, Spring passes the application context to `setApplicationContext()`.
- Postprocessor (before initialization): If there are any `BeanPostProcessors`, Spring calls their `postProcessBeforeInitialization()` method.

# **Bean Life Cycle**

- Initialize beans: If the bean implements InitializingBean, its afterPropertiesSet() method will be called. If the bean has a custom init method declared, the specified initialization method will be called.
- Postprocessor (after initialization): If there are any BeanPostProcessors, Spring calls their postProcessAfterInitialization() method.

# **Bean Life Cycle**

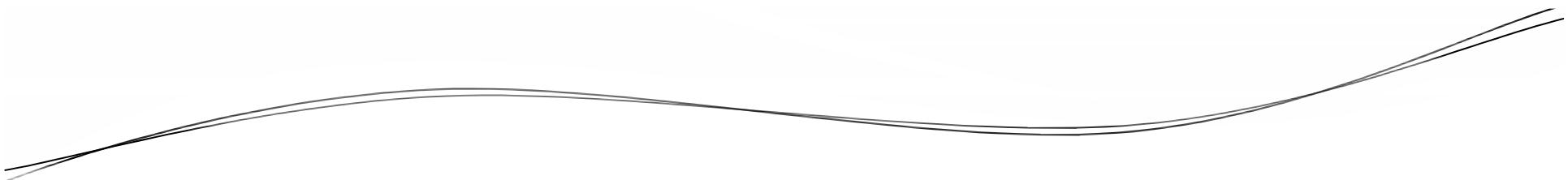
- Bean is ready to use. At this point the bean is ready to be used by the application and will remain in the bean factory until it is no longer needed.
- Destroy bean: If the bean implements DisposableBean, its destroy() method will be called. If the bean has a custom destroy-method declared, the specified method will be called.

# Injecting Properties through Constructors

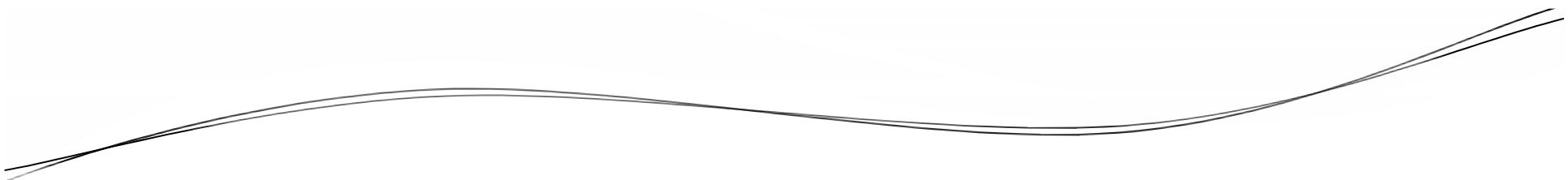
```
public class Employee {
 private String fname, lname;
 public Employee(String f, String l) {
 fname = f;
 lname = l;
 }
}
```

# Injecting Properties through Constructors

```
<bean id="emp"
 class="com.emp.Employee">
 <constructor-arg value="Joe"/>
 <constructor-arg value="Thomas"/>
</bean>
```

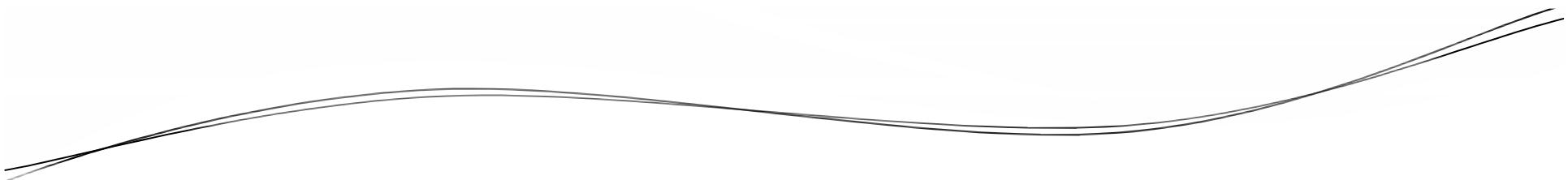


# Dependent Beans



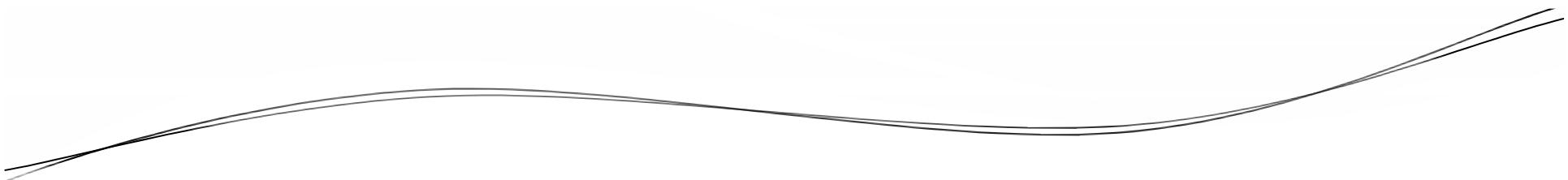
# **Dependent Beans**

- A bean is a dependency of another bean, is expressed by the fact that, one bean is set as a property of another.

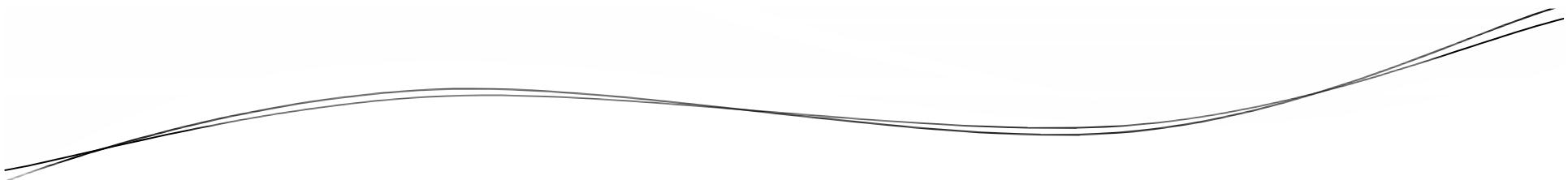


# **Dependent Beans**

- It is achieved with the `<ref/>` element or `ref` attribute in XML based configuration metadata of beans.

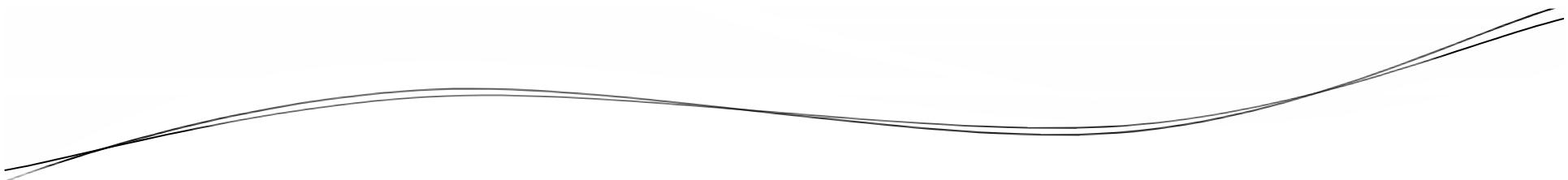


# **Bean Loading**



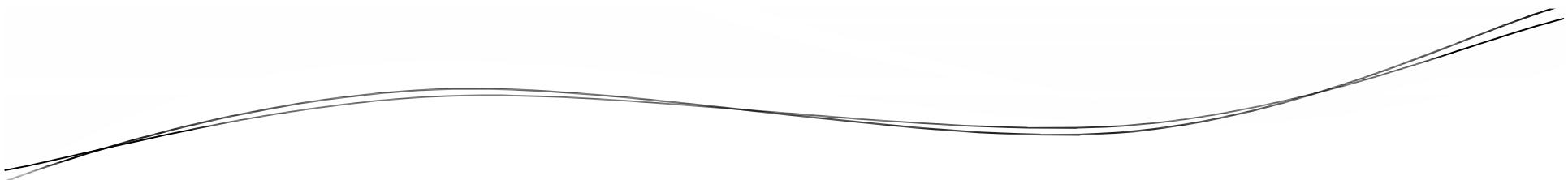
# Bean Loading

- In Spring, Bean Loading happens by 2 ways:
  - EAGER (DEFAULT)
  - LAZY



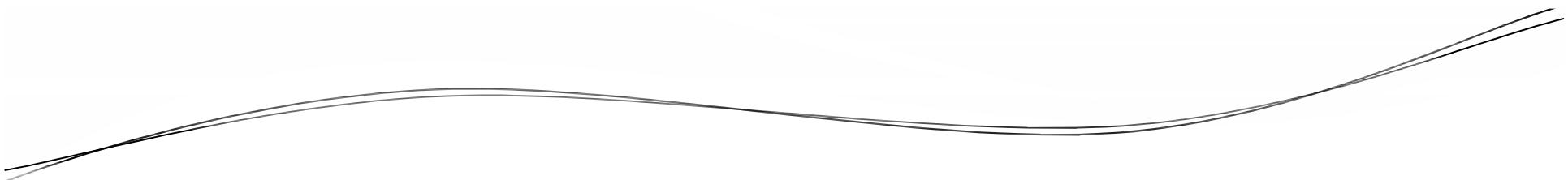
# Bean Loading

- The bean registered in the configuration unit gets instantiated as soon as the ApplicationContext is built.
- This is known as EAGER Loading.

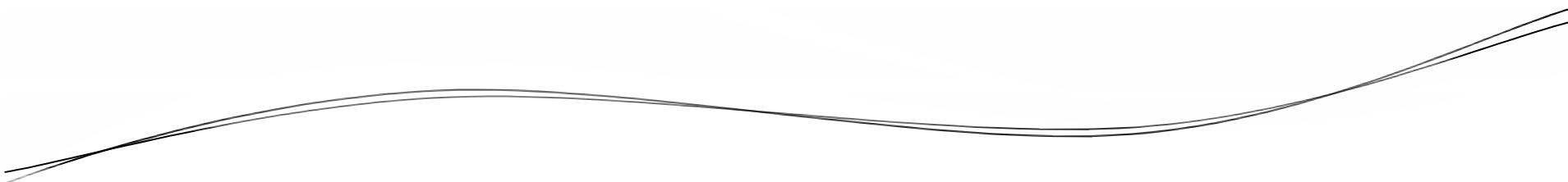


# Bean Loading

- The bean registered in the configuration unit gets instantiated only when the client program makes a request for the same.
- This is known as LAZY Loading.

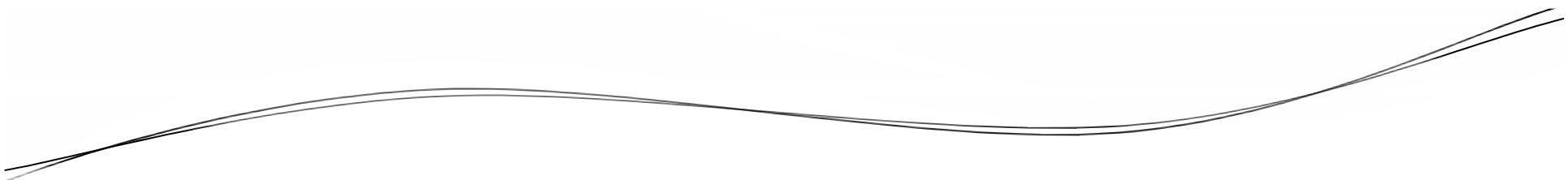


# AutoWiring



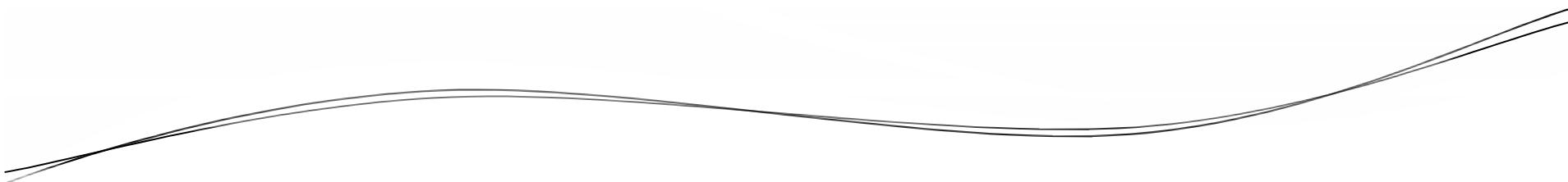
# AutoWiring

- Rather than explicitly wiring all of your bean's properties, you can have Spring automatically figure out how to wire beans.
- It is done by setting the **autowire** property.



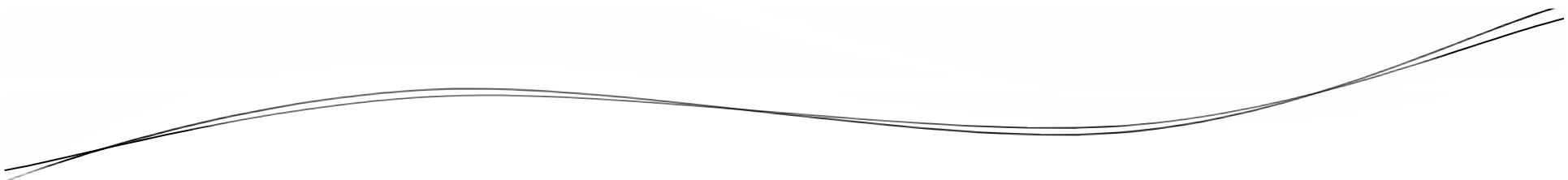
# AutoWiring

- Spring provides 3 types of autowiring:
  - byName
  - byType
  - constructor



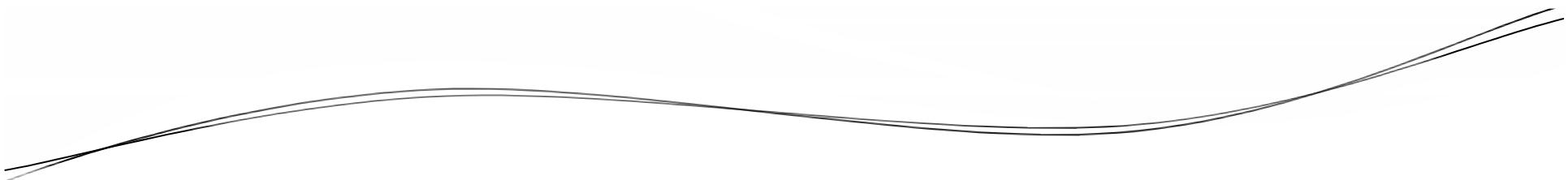
# AutoWiring

- `byName`
  - Attempts to find a bean in the container whose name is the same as the name of the property being wired.



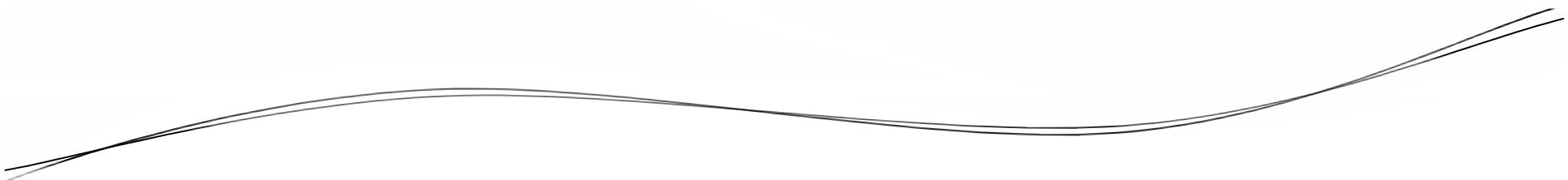
# AutoWiring

- byType
  - Attempts to find a single bean in the container whose type matches the type of the property being wired.

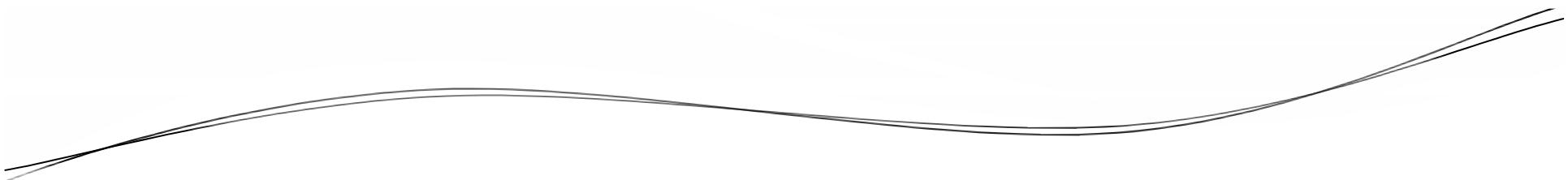


# AutoWiring

- constructor
  - Tries to match up a constructor of the autowired bean with beans whose types are assignable to the constructor arguments.

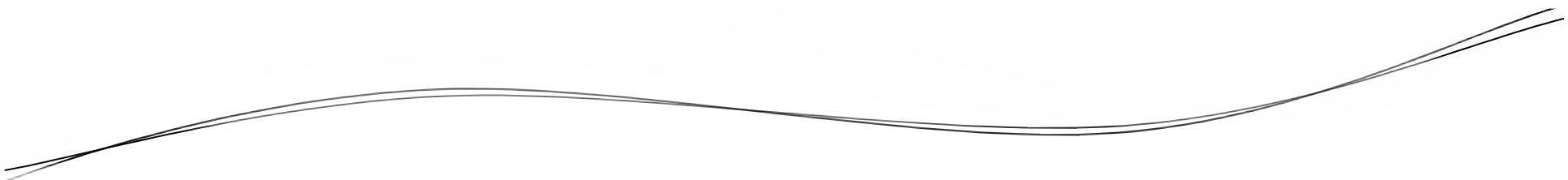


# **Bean Scopes**



# **Bean Scopes**

- Every bean registered in XML file has some scope.
- It is possible to modify scope of the bean using scope attribute of <bean> element.

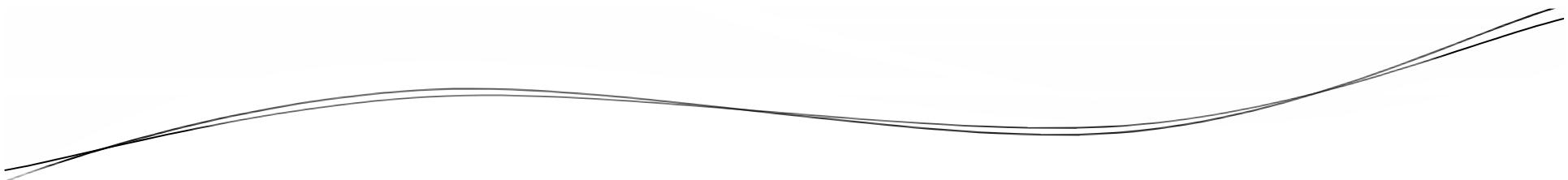


# **Bean Scopes**

- There are 5 different types of scopes:
  - singleton
  - prototype
  - request
  - session
  - global-session

# Bean Scopes

- singleton
  - It is the default scope.
  - Indicates that the bean configuration is singleton.
  - If the same bean is requested multiple times, spring returns the same object.



# Bean Scopes

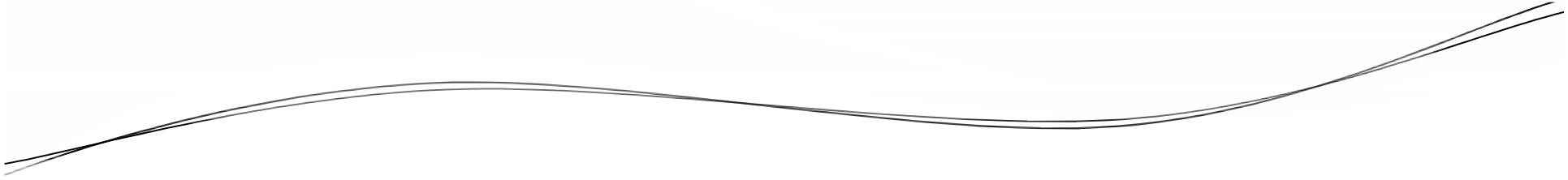
- prototype
  - Antonym of singleton.
  - If the same bean is requested multiple times, spring returns the a new object every time.

# Bean Scopes

- request
  - Applicable only in the context of Spring MVC.
  - The bean is alive until the response is generated.
  - For every new instance of HttpServletRequest, spring creates a new instance.

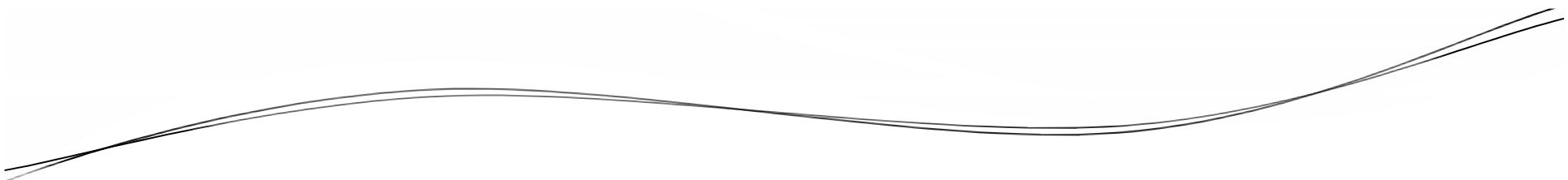
# Bean Scopes

- session
  - Applicable only in the context of Spring MVC.
  - The bean is alive until the session is over.
  - Bean can survive even if the response is generated.
  - For every new instance of HttpSession, spring creates a new instance.



# **Bean Scopes**

- global-session
  - Applicable in the context of Spring Portlet environment.
  - The bean is alive across multiple portlets.



# Let's Summarize

- Working with ApplicationContext
- Bean Life Cycle
- Understanding Dependency Injections
- Auto-wiring
- Bean Scopes