Sonali Sagar

Advanced Web Programming - Notes

*** Languages**
- **compiled**
  - compiler is required
  - compiles the whole source code into an executable
  - errors are detected at the time of compilation
  - compilation cannot proceed with error
  - faster
  - eg: c, c++
- **interpreted**
  - interpreter is required
  - interprets the code line by line
  - errors are detected at run time (executing the code)
  - execute can not proceed with error
  - slower
  - e.g.: HTML, JavaScript, (all scripting languages)

*** Applications**
- web applications (web site)
  - browser is compulsory
  - mainly HTML
  - browser can understand HTML
  - platform indenpendant
  - slower
  - eg: HTML
- **Native applications**
  - OS + CPU
  - mainly ASM language
  - CPU can understand only ASM
  - platform specific
  - faster
  - eg. C,C++

*** Web Application**
- HTML: design UI
- CSS: to add decoration in HTML code
- JavaScript: to add programming logic in HTML

*** HTML**
- ML: tags + data (content within tags)
- **tag:**
  - word enclosed in < and >
  - types
    - starting tag:
    - closing tag:
    - empty tag:
</br> =

- root tag:
  - also known as document type/tag/element
  - tag which starts a document and which ends the document

- **Parts**
  - head
    - title: shows the title on tab/window
    - meta: to add extra information (
    - style: to add CSS styles
    - link: to link external documents
    - script: to add JavaScript code
    - base: to decide the base url of the website
      - to configure the anchor tags
  - **body**
- Text Elements
  - inline
    - these elements do not create new line
    - eg: span, superscript etc
  - **block**
    - these elements create a new line at the end
    - eg: para, div, h1
- **Link elements**
  - anchor
- Table
  - head
  - body
  - footer
- **List**
  - ul
  - ol
  - definition list (dl)
- **Form**
  - input
  - select
  - textarea

*** Default Attributes**
- style: to add the CSS properties
- id: to identify the tag uniquely
- class: to identify similar type of element having same requirements

*** CSS**

- Cascading Style Sheet
- is introduced in HTML 4.0 to add decoration
- CSS3
- **Types**

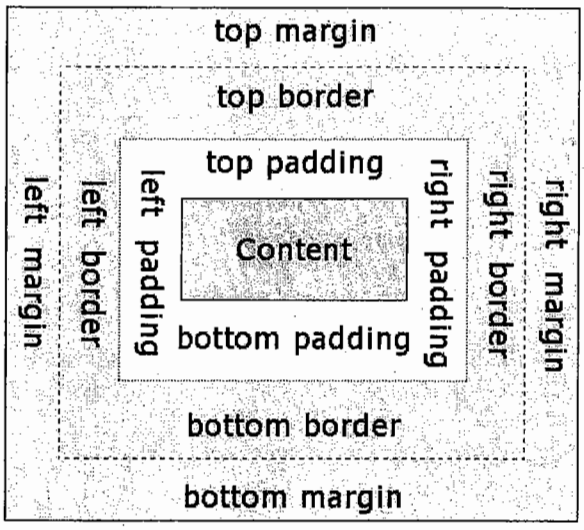
- inline style
  - to add decoration in a tag
  - **Disadvantage:**
    - repeat the same info for every tag
- **Internal Style**
  - to target multiple elements at once
  - internal to the current document
  - disadvantage:
    - repeat the same info for every page
- **External Style**
  - to target multiple pages at once
  - link tag is used to link external css file with the html document
  - eg: <link href=" <file name>"• type=" text/css"□ rel=" stylesheet"□ >
- browser default style
  - this style is by default applied on every html page
  - this style is browser dependant

- **Order**
  - browser default style: 1
  - external style: 2
  - internal style: 3
  - inline style: 4

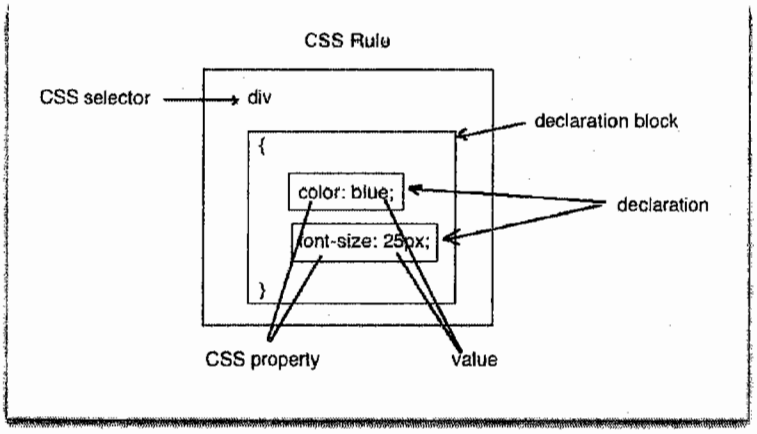
*** CSS selector**

- which can target multiple elements/tags at once
- types of selector
  - element selector
    - selects similar type of elements
    - e.g.: div{color:red;} -> all div elements will be in red color
  - group selector (,)
    - can select multiple type of elements
    - e.g.: div, p {"¦} -> all div and para will be targeted
  - id selector (#)
    - to target only a specific tag (having specific id)
    - eg: div#specialdiv {..}
  - class selector (.)
    - to target elements having same class name
    - eg: div.greendiv {..}
  - pseudo selector: (:)
    - which gets executed in a specific condition
    - eg:
      - :hover -> when mouse pointer is on top of this element
      - :nth-child -> for n-child

CSS Box Model →



CSS Rule →



***JavaScript**

- is a interpreted language
- browser acts as interpreter
- is a object based language
- usage:
  - to add programming logic in HTML
  - these days it is also used by third party software like Unity3D to add programming logic in mobile games
  - these days it is also used on server to communicate with Web server to create dynamic web pages

*** Variables**

- data type can NOT be assigned in JavaScript
- JS decides the data type by looking at the variable"™s CURRENT value
- variables can be redeclared

- eg:

```
name = "sunbeam" ;  // string
salary = 6.7;       // number
count = 10;         // number
canVote = true;     // boolean
```

- **Scopes**
  - Global scope
    - create a variable without using var keyword inside a function
    - or declare a variable outside any function
    - e.g.: name = "sunbeam" ;
  - Local scope
    - declare a variable with var keyword inside a function
    - e.g.: var myname = "sunbeam" ;
  - var keyword is used to declare a variable
    - if var is used inside a function: it creates local var

```
function function1() {

var myname = "test" ; // local
}
```

    - if var is used outside a function: it create global var

```
var myname = "test" ; // global

function function1() {
}
```

- **function**
  - function is a keyword used to declare a function
  - syntax:

```
function <function name> (<number of parameters>) {
    // function body
}
```

  - a function can be called by passing variable number of parameters
    - if less number of params are passed: then remaining will be assigned undefined value
    - if more number of params are passed: then extra params will be ignored

```
function add(num1, num2) {
  console.log("num1 = " + num1);
  console.log("num2 = " + num2);
```

```
}

add();       // num1 = undefined, num2 = undefined
add(10);     // num1 = 10, num2 = undefined
add(10, 30); // num1 = 10, num2 = 30
add(1,2,3,4); // num1 = 1, num2 = 2: rest of the params will be ignored
```

- **JS pre-defined words**
  - undefined:
    - a var is declared but not defined
    - the value is not set
  - NaN:
    - not a number
    - mathematical operations are performed on a non-number value
    - eg: undefined + undefined = NaN
      undefined + 10 = NaN
      "test" * "test" = NaNs

- **Browser**
  - special application which can understand HTML
  - it is used to render/display html pages
  - **components**
    - UI component
    - data storage
      - store data
      - cookies
      - HTML5
        - session storage
        - local storage
    - networking component
    - javascript engine
    - rendering engine
      - Apple Safari: webkit
      - Google Chrome: webkit -> Blink
      - Mozilla Firefox: Gecko
      - Opera: gastro
      - IE: -

*** URL**

- Uniform Resource Locator
- to identify any resource from internet uniquely
- contains components
  - scheme: (http)
    - generally known as protocol

- decided how to fetch the resource from internet
- eg: http, https, file, ftp, tftp etc
- domain information: (www.mywebsite.com)
- www: sub-domain
- mywebsite: domain name (gets resolved into a unique IP address by DNS)
- com: top level domain
- port number: (80)
- identifies the service running on the server
- if port number is not provided then the default port number for specified scheme is used
- eg: http:80, https:443, ftp:21
- file / path: (myfolder/myfile.html)
- path to the resource
- query string: (firstname=test&seconname=test)
- is the way to provide input to a page
- it always present in key=value format
- e.g.: firstname=test and secondname=test
where: firstname, secondname are keys
test, test are values
- multiple key-value pairs are separated by &
- hash (#) component: (#top)
- used to link a part of same document

eg: http://www.mywebsite.com:80/myfolder/myfile.html?firstname=test&seconname=test
eg: http://www.mywebsite.com:80/myfolder/myfile.html#top

*** Object**

- collection of different data members (properties) and functions associated with them
- **2 ways to create an object**
- by using new Object
- Object is a kind of template given by JavaScript
- eg:

```
var person = new Object();    <- Object
person.name = "test" ;        <- property
person.address = "Karad" ;    <- property
console.log("name : " + person.name);

var person1 = new Object();
person1["name"] = "test1";
person1["address"] = "Pune";
console.log("name : " + person["name");
```

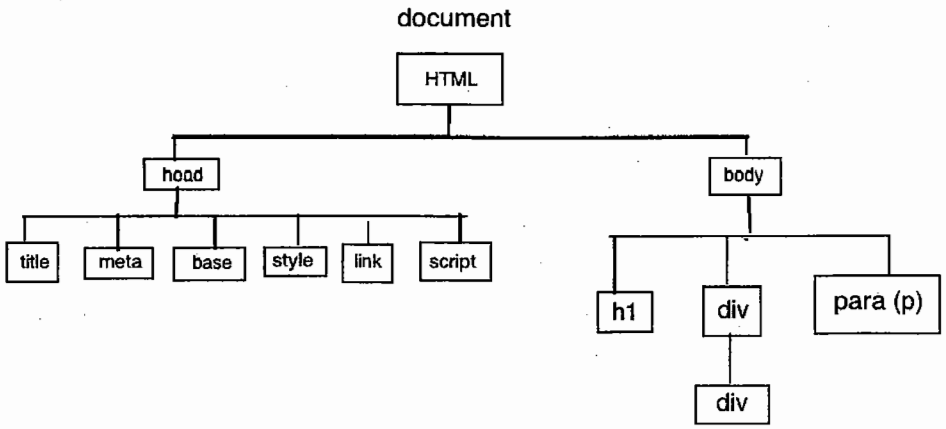
- json
- JavaScript Object Notation
- the simplest way to create an object

- these days, json is used to transfer data from one entity to another
  - used to send data from server to client (mobile Facebook application)
  - because
    - json is light weight text file
    - easy to parse (extracting information) json response
- eg:

```
var person = {
    "name" : "test" ,
    "address" : "karad"
};
console.log("name : " + person.name);
```

- DOM

- Document Object Model



- hierarchy of tags
- W3C has 3 types
  - Core DOM
    - language independent
  - HTML DOM
  - XML DOM

- JavaScript built in objects

- these objects are created by browser at the time of loading the page
- document
  - representation of HTML DOM
    - using DOM object (document), html tags can be accessed/modified at run time
  - it contains JS object of every html tag inside the html file
  - using document any html tag can be accessed by using:
    - getElementById(): returns a tag which has the same id as that of the input
    - getElementsByTagName(): by matching the tag names
    - getElementsByName(): by matching the name attribute

- getElementsByClassName(): by matching the class name
- every tag contains property named innerHTML which can be used to set / get the contents of that tag except input (value)
- sessionStorage:
- localStorage:
- navigator:

***** HTML5 functionality *****

*** Web Storage**
- used to store data temporarily or permanently
- key-value pairs can be stored in side web storage
- only ASCII character (string) can be stored in web storage
- implemented and maintained by browser internally
- stored in different location for different browser
- **Types**
  - session storage
    - stores the data temporarily
    - till the session is alive
    - session:
      - the duration of browser
      - it gets created at the time of opening browser and gets closed at the time of closing browser
  - local storage
    - data gets stored permanently
    - data upto 5MB can be stored in local storage

*** Geolocation**
- to get the current location in terms of Latitude and Longitude
- ways (providers) to retrieve the current position
  - GPS hardware
  - IP address
  - cell triangulation (A-GPS)

*** Multimedia**
- video

*** Canvas**

*** Pop up box**
- alert
  - used to display readonly message
  - e.g. alert("hello..”•);
- prompt
  - used to get a single line input from user
  - e.g. var name = prompt("enter name..”•);
- confirm

- used to get confirmation from user
- e.g. var answer = confirm("are you hungry?"); // true or false

*** Date object**

*** timer functions**
- used to perform functionality after an interval
- setInterval()
  - used to call a function every after some interval
  - setInterval(<function name>, <interval in mili-seconds>);
- clearInterval()
  - stops calling a function which is started by setInterval()
- setTimeout()
  - used to call a function only once after few mili-seconds
  - e.g. setTimeout(<function name>, <interval in mili-seconds>);
- clearTimeout()
  - stops calling the function which is started by setTimeout()

*** jQuery**
- javascript library of commonly used functions
- benefits
  - free and open source
  - has a very good community

*** XML**
- is a Markup Language
  - made up of elements and data
- eXtended Markup Language
- standardized language by w3c
- imp
  - the elements are user-defined: xml writer must define the xml elements
  - simple text file format
  - xml is interpreted
  - an interpreter is require: by default browser is used to interpret the xml
- usage
  - structuring the data
  - to transfer data from one entity to another in commonly understandable format
  - to store very small amount data
- xml vs html
  - html is used for designing user interface for browser
    xml is NOT used to design UI
  - html has pre-defined tags
    xml has user-defined elements
  - html is case in-sensitive : <table></Table>
    xml is case sensitive

*** XML element**

- word enclosed in < and >
- types
  - starting element: <name>
  - closing element: </name>
  - empty element
    - normal syntax: <name></name>
    - shorthand: <name />

  root element
  - which starts and closes the document
  - also known as document type or document element
- rules to create an element
  - every file must have one and only one root element
  - Special characters like space can not be used in element
    - <First Name> <- incorrect element
    - <FirstName>, <First_Name>
  - only _ can be used in element
    - <First_Name>
  - elements cannot start with number
    - <1Name> <-- incorrect element-->
    - <Name>
  - every starting element must be closed with closing element
    - <Name>test <- incorrect -->
    - <Name>test</Name>
  - element name is case sensitive
    - <Name> and <name> are two different element

*** Attribute**

- it adds more information in xml element
- it is presented in name=value pair
  - e.g. phoneType="work"
- rules
  - can not be repeated
    - e.g. <Phone phoneType="work" phoneType="special"></Phone>
      <Phone phoneType="work"></Phone>
  - in name=value format
  - can NOT have multiple values at the same time
    - e.g. <Phone phoneType="work","special"></Phone>
  - can NOT have a child attribute
- attribute-only element
  - having only attributes
    - e.g. <Person name="test" address="Karad" />
- mixed type element
  - has both: child element and attribute
  - <Person name="test" address="Karad">
    <Phone phoneType="special">4544</Phone>
    <Phone phoneType="work">4544</Phone>

<Phone phoneType="personal">45464</Phone>
</Person>

*** Parts**
- XML Header
  - <?xml version="1.0" encoding="UTF-8" ?>
  - it is optional
  - where:
    - version: specification version used to create this xml
    - encoding: type of characters using in this xml
      - UTF: Unicode Transformation Format
        - UTF-8: 8 bit or 1 byte
        - UTF-16: 16 bit or 2 bytes (wide character)
- XML body
  - collection of user defined elements

*** CDATA**
- the characters included in the CDATA section are not parsed
- special meaning of these characters will not be used while presenting xml
- <![CDATA[> <data> <]]>

* Well formed xml document
- the xml is following all the syntactical rules specified by W3C

*** Valid XML**
- a well formed xml which follows all the logical rules as well
- logical rules can be defined by using DTD or XML Schema
- every Valid xml must be well formed xml but every well formed xml may NOT be valid xml

*** DTD**
- Document Type Definition
- used to define the logical rules
- types
  - a root element (!DOCTYPE)
    - used to defined a root element
    - <!DOCTYPE <element name> [<logical rules>]>
  - elements
    - simple element (!ELEMENT)
      - <!ELEMENT <element name> (#PCDATA)>
      - where
        - #PCDATA: all parcelable characters
    - element having child element(s) where order is imp
      - <!ELEMENT <element name> (<child element1>, <child element2>...)>
      - element is made up of sequence of child elements
      - child elements must be present in the pre-defined order
    - element having child element(s) where order is not imp
      - <!ELEMENT <element name> ANY>

- where
  - ANY: any element in any order
- empty element
  - element having no data
  - <!ELEMENT <element name> EMPTY>

- **Wild characters**
  - *: element can appear zero or more times
  - +: element can appear one or more times
  - ?: element can either appear only once or absent

- **attributes**
  - !ATTLIST is used to define an attribute
  - <!ATTLIST <element name>
    <attribute name> CDATA <type> >
  - e.g.

```
<!ATTLIST Phone
    phoneType CDATA #IMPLIED>
```

where
- type = #IMPLIED -> optional -> ignore
- type = "value" -> default value -> default value
- type = #REQUIRED -> compulsory

*** DTD**
- Types
  - internal DTD
    - the DTD definition written in the same xml file
    - the file becomes cluttered with xml content and DTD content
  - external DTD
    - a new file with .dtd extension must be created with all the definitions
    - external DTD can NOT start with DOCTYPE
    - to load external DTD SYSTEM keyword is used
    - <!DOCTYPE <element name> SYSTEM "<dtd file name>">
  - mixed DTD
    - combination of both internal and external DTD
    - <!DOCTYPE <element name> SYSTEM "<dtd file name>" [
      <internal DTD rules>
      ]>

- **Limitations**
  - DTD has its own syntax
  - DTD can not differentiate between character and number
  - values can not be restricted using DTD
  - DTD can not add any restriction on type of data and number of times it gets repeated

- DTD can not understand the xml namespace

*** xml namespace**
- group which separates multiple elements having same logical structure
- xmlns keyword is used
  - syntax:
    <table xmlns:<prefix>="http://<namespace name>"></table>
- to put an element inside a namespace
  - <prefix:<element name>>
  - e.g. <p1:table>..</p1:table>
- rules
  - every namespace can be used inside the same element or child elements where it is defined
  - by default all the child elements are the part of same namespace of their parent element
  -

*** XML Schema**
- xml schema is another way to define logical rules
- data types
  - string: accepts any character including number and special symbols
  - integer: positive and negative
  - decimal: float values
  - positiveInteger:
  - negativeInteger:
  - date:
  - time:
  - dateTime:
  - positiveDecimal:
  - negativeDecimal:
- custom types
  - add restrictions

- element Types
- Simple element
  - does NOT contain any child element
  - does NOT contain any attribute
- Complex element
  - may contain at least a single child element
  - may contain at least a single attribute

- rules
- xml schema must be written outside the xml file (externally)
- the file extension must be .xsd (xml schema definition/document)

- Element
- use: xs:element
- syntax:
  <xs:element name="<name of element" type="<element type>" />

- to create custom type

```
<xs:element name="<element name>" >
<xs:simpleType>
  <xs:restriction base="<data type>">
      <!— add the required condition —>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

- types
  - simple element

```
<xs:element name="<name of element>" type="<element type>" />
```

  - element having child element(s) in a specific order
    - all the child element(s) must be present

```
*  <xs:element name="<element name>" >
<xs:complexType>
  <xs:sequence>
      <!— child element(s) —>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

  - element having child element(s) where order is not imp
    - all the child element(s) must be present

```
*  <xs:element name="<element name>" >
<xs:complexType>
  <xs:all>
      <!— child element(s) —>
    </xs:all>
  </xs:complexType>
</xs:element>
```

  - element having child element(s) where only one child element must be present

```
*  <xs:element name="<element name>" >
<xs:complexType>
  <xs:choice>
      <!— child element(s) —>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

- element having repeating child element
  - occurrence indicator
    - minOccurs: minimum number of times the element should me present
      - = 0: element becomes optional
    - maxOccurs: maximum number of times the element should me present
      -> unbounded: infinity

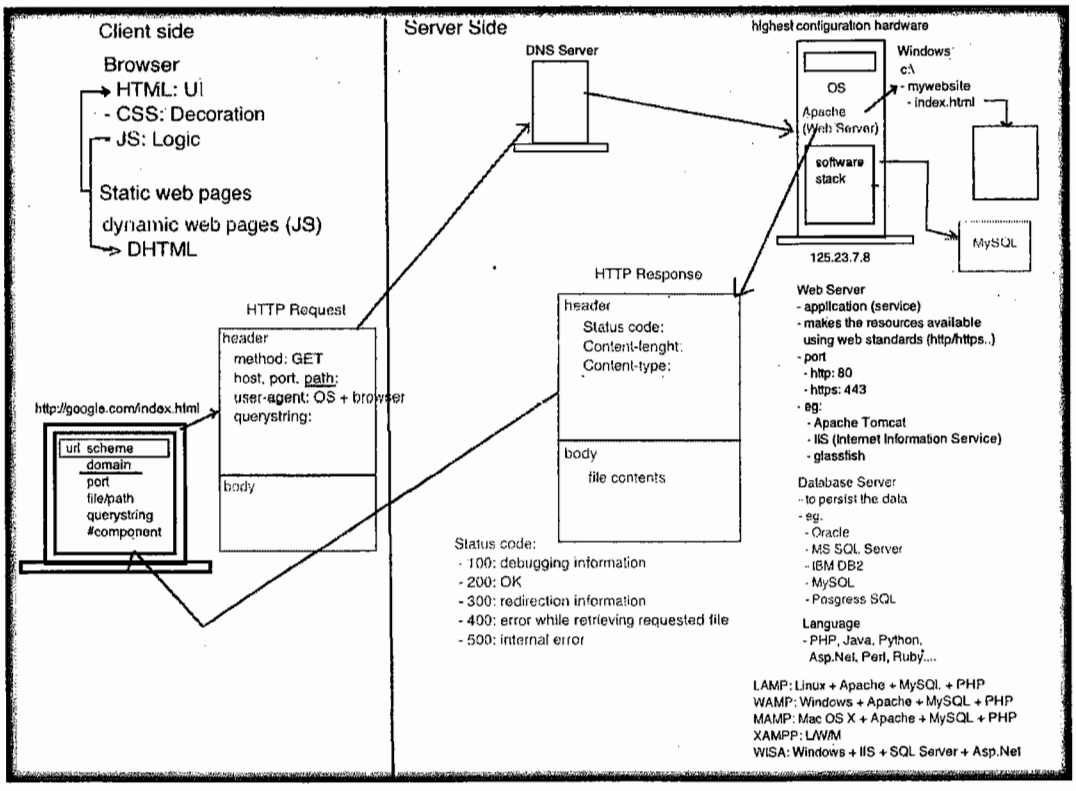
- element having an attribute

*** Path**

- File system path
  - Absolute path
    c:\windows\calc.exe
  - Relative path
    c:\ -> windows\calc.exe
    c:\windows -> calc.exe

*** PHP**

Client – Server Architecture →



- Personal Home Page
- PHP: Hypertext Preprocessor
- **Features**
  - using PHP, dynamic web pages can be designed

- PHP is a scripting language
- PHP is a interpreted language

- Why PHP
  - its very simple to learn and use
  - Open source and free
  - it can be used to design huge web site (having many pages)
  - supports wide range of Database like MySQL, Oracle, SQL Server etc

*** Language Features**

- code must be written inside <?php ?>
  - if page contains only PHP code then ?> is optional
  - if page contains both PHP as well as HTML then ?> is compulsory
- Syntax
  - all the keywords, function names and class names are case in-sensitive
  - variable names are case sensitive
- Variables
  - variable name must start with $
    - $name = "sunbeam";
  - variable can not decide its data type
  - php decides the data type by looking at the current value
- data types
  - string:
    -
  - int: (integer) -> whole numbers
  - float: decimal numbers
  - bool: boolean (true/false)
  - Array:
  - NULL:
  - object: instance of a class
- Array
  - collection of similar or dis-similar data type
  - syntax: $array = array(<value1>, <value2>….);
  - count(): returns the length of array
  - types
    - single dimensional
    - multi-dimensional
    - associative array
- function
  -
- scope
  - local
    - declared inside function
  - global
    - declared outside of any function
    - global variables can NOT be used directly inside function
      - global keyword must be used to use a global variable inside a function

```
$num = 100;
    function function1() {
        global $num;        ///  <- global keyword is required here
        echo "Num = $num";
    }
```

- static
  - that retains it value

*** Superglobals**

- associative array
- $_GET:
  - array which holds all the input parameters sent using GET method
- $_POST:
  - array which holds all the input parameters sent using POST method
- $_REQUEST:
  - holds all the input parameters present in Request
- $_SERVER:
  - holds parameters sent through HTTP Request

Annotations

- Added in Java 5.0

Inheritace

- Use of class
  - Reusablitiy -- fields & methods.
  - Runtime polymorphism -- Override methods.
- Use of abstract class
  - Reusablitiy -- fields & methods.
  - Runtime polymorphism -- Override methods.
  - Contract -- force to override abstract methods.
- Use of interface
  - Runtime polymorphism -- Override methods.
  - Contract -- force to override all methods.
  - Used to develop/provide "specifications".
    - e.g. JDBC, Servlet, EJB, ...
- Use of marker interface
  - To mark/tag a type with some functionality. It associate extra info (metadata) with the type.
  - e.g. Serializable, Cloneable, ...
    - Serialiable --> Allows JVM to convert object into sequence of bytes and vice versa -- in ObjectOutputStream.writeObject() and ObjectInputStream.readObject().
    - Cloneable --> Allows JVM to create a copy of the object into Object.clone() method.
- Annotations used to associate metadata with a type, a field, a method, a method parameter, ...

Annotations types/retention policies

- There are three annotation retention policies
  - SOURCE
  - CLASS
  - RUNTIME

SOURCE

- Processed by the compiler & discarded.
- Not added in .class file.
- e.g. @Override --> tells compiler to compare method decl with super class method decl. If not matching, raise compiler error.
- e.g. @SupressWarning --> tell compiler to remove few warning.
- e.g. @Deprecated --> tell compiler to raise warning if this type/member is used.

CLASS

- Processed by the class loader & then discarded.
- Not present in JVM at runtime.
- Mainly used for config/settings of a type.

RUNTIME

- Loaded & maintained at runtime in JVM memory.

- Can be processed using reflection at runtime using cls.getAnnotations().
- Heavily used by all frameworks e.g.
  - Hibernate: @Table, @Column, @Entity, @OneToMany, ...
  - Spring: @Controller, @RequestMapping, @PathVariable, @ModelAttribute, ...

Custom annotation

- step 1. Declare annotation type

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
public @interface Readme {
	String value(); // developer -- default field
	String company() default "Sunbeam";
}
```

- step 2. Apply annotation type

```
@Readme(value="Nilesh", company="Sunbeam Infotech")
class A {
	// ...
}
@Readme(value="Nilesh")
class B {
	// ...
}
@Readme("Amit")
class C {
	// ...
}
```

- Access annotation using reflection

```
Annotation[] anns = cls.getDeclaredAnnotations();
for (Annotation a : anns) {
	System.out.println("Annotation : " + a);
	if(a instanceof Readme) {
		Readme r = (Readme) a;
		System.out.println("Company : " + r.company());
		System.out.println("Developer : " + r.value());
	}
}
```

Java Dynamic Proxies

- Feature is added in Java 1.3 -- Dynamic Proxies.
- Proxy objects are used to give access to core business logic indirectly.
- It can be used for
  - Modifying args before call to method
  - Modifying result after call to method
  - Restricting access to the method
  - Logging access to the method
  - Implement transaction management
- All these additional functionalities (cross-cutting concerns) can be implemented without modifying core business logic.
- This programming paradiam is called as "AOP".

Implemening proxies

- step 1. Implement interface & implementor class

```
// AddSubtract.java
package com.sunbeaminfo.sh;

public interface AddSubtract {
	int add(int a, int b);
	int subtract(int a, int b);
}
// AddSubtractImpl.java
package com.sunbeaminfo.sh;

public class AddSubtractImpl implements AddSubtract {
	@Override
	public int add(int a, int b) {
		return a + b;
	}

	@Override
	public int subtract(int a, int b) {
		return a * b;
	}
}
```

- step 2. Implement handler class

```
// AddSubtractProxyHandler.java
package com.sunbeaminfo.sh;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class AddSubtractProxyHandler implements InvocationHandler {
	private AddSubtract implObj;
	public AddSubtractProxyHandler(AddSubtract implObj) {
		this.implObj = implObj;
	}

	public static AddSubtract getProxy(AddSubtract implObj) {
		try {
			return (AddSubtract) Proxy.newProxyInstance(
					AddSubtract.class.getClassLoader(),
					new Class[] { AddSubtract.class },
					new AddSubtractProxyHandler(implObj)
				);
		} catch (Exception e) {
			return null;
		}
	}

	public static AddSubtract getProxy() {
		return getProxy(new AddSubtractImpl());
	}
```

```
	@Override
	public Object invoke(Object proxyObj, Method method, Object[] args) throws
Throwable {
		System.out.println("Calling Method : " + method);
		Object ret = method.invoke(implObj, args);
		// post-processing
		System.out.println("Called Method : " + method);
		return ret;
	}
}
```

- step 3. Invoking methods

```
// ProxyMain.java
package com.sunbeaminfo.sh;

public class ProxyMain {
	public static void main(String[] args) {
		AddSubtract obj2 = AddSubtractProxyHandler.getProxy();
		System.out.println("Type : " + obj2.getClass().getName());
		int res2 = obj2.add(12, 5);
		System.out.println("add : " + res2);
		System.out.println();
	}
}
```

Java Build Tools

- Java build tools are used to build huge java projects.
- They also does following:
  - Add required jars & their dependencies in project classpath & deployment assembly.
  - Compilation of the project.
  - Testing & packaging project deployment assembly.
  - Deploying the project.
- Few popular java build tools are
  - Ant
  - Maven
  - Gradle

Maven

- Maven version: 3.5+

Maven Installation

- Native installation:
  - Ubuntu: sudo apt-get install maven
  - "mvn" command is used to perform build or other tasks.
    - e.g. mvn -v (to check version)
- Eclipse: Eclipse Mars+ have built-in maven.
  - Maven 3.5

- m2e plugin --> Maven 2 Eclipse plugin.

pom.xml: Project Object Model

- pom.xml file should be kept in project root directory.
- pom.xml contains build config/settings of project.
  - basic config: java version, other libs versions
  - dependencies: third party jar files to be added in project.
  - packaging info: jar or war, name, groupId/artifactId
  - repositories: custom repositories
  - build settings: if different than the default one.

Life cycles

- Maven have three life cycles.
  - build: build the project with multiple phases.
  - clean: clean the project (delete generated files .class, ...)
  - site: build documentation of the project.
- For example
  - mvn clean
    - delete generated files .class and other files.

Build phases

- Maven build life cycle have multiple phases.
- These phases are followed in sequence.
- Executing any phase will ensure that all phases above that are executed.
- Maven build life cycle phases
  - validate: project validation & download dependencies.
  - compile: compile project (.class).
  - test: runs all unit test cases.
  - package: create .jar or .war file.
  - install: install jar in local repo.
  - deploy: install jar in remote repo.
- "compile" is default build phase.

Repositories

- Maven repository is collection of standard/third-party jars.
- There are three types of repositories.
  - Local repository
  - Remote repository
  - Central repository

Local repository

- Available on each development machine.
- The default path is $HOME/.m2 directory on Linux; while C:\Users\username.m2 on Windows.
- This directory can also be specified using environment variable M2_HOME.
- It contains
  - settings.xml contains default maven settings
  - "repository" directory in which jars are arranged in hierarchical order.

Remote repository

- Usually setup by organizations for hosting the required jars, which include third party jars + other project jars.
- The path of remote repository need to be added in pom.xml

Central repository

- Maintained by maven on their server i.e. https://mvnrepository.com/repos/central

Creating Maven Project in Eclipse

- step 1. File -> New -> Maven Project
  - Check-mark -- Create simple project -- Next.
  - Enter -- groupId, artifactId, project name, packaging (jar/war).
- step 2. For web project
  - Select packaging as "WAR" (in above step).
  - Project --> right click --> Java EE tools --> Generate deployment descriptor stub.
- step 3. pom.xml -- add versions & dependencies

```
<properties>
	<maven.compiler.source>1.8</maven.compiler.source>
	<maven.compiler.target>1.8</maven.compiler.target>
	<java.version>1.8</java.version>
</properties>
<dependencies>
	<dependency>
		<groupId>javax.servlet</groupId>
		<artifactId>javax.servlet-api</artifactId>
		<version>4.0.0</version>
		<scope>provided</scope>
	</dependency>
	<dependency>
		<groupId>javax.servlet.jsp</groupId>
		<artifactId>javax.servlet.jsp-api</artifactId>
		<version>2.3.0</version>
		<scope>provided</scope>
	</dependency>
	<dependency>
		<groupId>javax.servlet</groupId>
		<artifactId>jstl</artifactId>
		<version>1.2</version>
	</dependency>
	<dependency>
		<groupId>javax.el</groupId>
		<artifactId>javax.el-api</artifactId>
		<version>3.0.0</version>
	</dependency>
	<dependency>
		<groupId>mysql</groupId>
		<artifactId>mysql-connector-java</artifactId>
		<version>8.0.16</version>
	</dependency>
</dependencies>
```

- step 4. After changing basic project settings like java version, maven project should be updated.
    - Project -> right click -> Maven -> Update project.

Maven directory structure

- src/main/java
    - Source code of project (.java files in packages)
- src/main/resources
    - Resources of project (.xml, .properties, ...)
- src/test/java
    - Source code of Unit test cases (.java files in packages)
- src/test/resources
    - Resources for Unit test cases (.xml, .properties, ...)
- src/main/webapp
    - Web pages (.html, .jsp)
    - Web resources (.js, .css, .jpg, ...)

Maven project compilation

- Project -> right click -> Run as
    - Maven clean
    - Maven install
- Maven install
    - Execute all phases i.e. validate, compile, test, package & install.

Maven Parent/Super POM

- Parent pom.xml is kept in one of the following locations.
    - parent directory of child project.
    - local repository.
    - remote repository.
    - relative path to current project.
- Child pom.xml
    - contains <parent> tag to point to parent pom.xml.
- Child project inherit all settings of parent. If modified, child settings override parent settings.
- Major advantage of parent POM is that, certain settings common to all projects need not to be repeated in each child project. Versions of jars will be availble to child pom.

JDBC

- Stands for Java DataBase Connectivity.
- To deal with RDBMS (not NoSQL).

JDBC specification

- Specification is given in form of interfaces by "Sun Microsystems".
- The important interfaces (java.sql package) are as follows:
    - Driver: Used to create and manage database connections.
    - Connection: Encapsulate socket connection which communicate with database.
    - Statement: Encapsulate SQL query to be executed on db.

- ResultSet: Represents database response to the query (rows and cols), used to fetch result row by row.
- PreparedStatement: Used to execute parameterized query on the database.
- CallableStatement: Used to execute stored procedure on the database.

- The specification is implemented in form of JDBC driver.
  - Internally each JDBC driver contains set of classes inherited from above given interfaces.
  - This ensures that same methods are present in all driver implementations.
- Thus JDBC program remains same irrespective of which JDBC driver or database you are using.

JDBC Driver

- JDBC Driver is a special program used to convert java requests to database understandable form and database response to java understandable form.
- There are four types of JDBC drivers.
  - Type I
  - Type II
  - Type III
  - Type IV

Type I driver

- Also called as "JDBC ODBC Bridge Driver".
- ODBC drivers are implemented in C and can connect to older databases like Foxpro, Access, etc.
- Java has built-in JDBC driver to communicate with ODBC driver which in turn work with database.
- Name of driver class is sun.jdbc.odbc.JdbcOdbcDriver.
- Due to multiple layers, it is slower.
- Also ODBC support is not fully present on all platforms.

Type II driver

- Partially implemented in C and partially in Java.
- Use JNI to communicate between Java and C layer.
- Better performance than Type I.
- Different driver for each database.
- Not truly portable (as partially in C) and Outdated.
- e.g. Oracle OCI driver

Type III driver

- JDBC driver communicate with a middleware, which in turn communicate with actual database.
- e.g. Weblogic RMI driver

Type IV driver

- Completely implemented in Java and hence truly portable.
- Different for each database.
- Internally use sockets to communicate with database.
- e.g. Oracle Thin Driver.

JDBC Programming Steps

- step 1. Add JDBC driver into CLASSPATH
  - On command line:
    - WINDOWS: set CLASSPATH=/path/of/file/driver.jar;%CLASSPATH%
    - LINUX: export CLASSPATH=/path/of/file/driver.jar;$CLASSPATH
    - javac -d . ClassName.java
    - java pkg.ClassName
  - In Eclipse:
    - Right click on project, Add New Folder. Give name "lib".
    - Copy driver jar file into "lib".
    - Right click on project, go to Properties -> "Java Build Path" -> "Libraries" -> "Add Jars". Now select driver jar from "lib" directory within current project.
- step 2. Load and register driver class
  - Class.forName(DB_DRIVER);
    - DB_DRIVER = "com.mysql.jdbc.Driver" OR "oracle.jdbc.OracleDriver".
- step 3. Create connection
  - Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASS);
    - DB_URL = "jdbc:mysql://server:3306/test" OR "jdbc:oracle:thin:@server:1521:oracle"
- step 4. Create Statement
  - Statement stmt = con.createStatement(); // OR prepareStatement();
- step 5. Execute query
  - Non-Select query: int cnt = stmt.executeUpdate(sql);
  - Select query: ResultSet rs = stmt.executeQuery(sql);
    - Process ResultSet to fetch rows & then close it.
- step 6. Close statement & connection
  - stmt.close();
  - con.close();

DriverManager and Connection

- Driver registration
  - In older drivers, driver registration was done manually as follows.
  - Class c = Class.forName("pkg.DriverClassName");
  - Driver drv = (Driver)c.newInstance();
    DriverManager.registerDriver(drv);
  - Now, driver class is automatically registered, when loaded for first time (due to its static block).

    Class c = Class.forName("pkg.DriverClassName");
- In JDBC 4.0, even this step is optional, because JDBC driver implementation declare service providers in META-INF/services/java.sql.Driver file within driver jar. These classes are automatically loaded by DriverManager during initialization.
- The registration process keep track of which JDBC driver is suitable for which RDBMS. This driver is selected to create JDBC connection for that database.
- Connection Creation
  - con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASS);
  - DriverManager internally search for appropriate driver object (already registered) and then call its connect() method to create actual JDBC Connection and return it.
  - Connection object encapsulate the socket connection with the database. The query execution and result fetching is done from the same socket connection.

JDBC Statements
Statement

- Need to create SQL query by string concatenation (for user inputs). This may make applications prone to SQL injection.

SQL injection

- SQL injection is one of the most common web hacking techniques.
- It is the placement of malicious code in SQL statements, via web page input.
- If not handled, it may destroy the whole database.
- String empno = sc.nextLine();
  String sql = "DELETE FROM EMP WHERE EMPNO=" + empno;
- If user input is "11"
  - Query: DELETE FROM EMP WHERE EMPNO=11
- If user input is "11 OR 1"
  - Query: DELETE FROM EMP WHERE EMPNO=11 OR 1

PreparedStatement

- PreparedStatement is inheried from Statement.
- It is used to create parameterized queries. These queries contains placeholder (?) for the values.
- These are most commonly used to prevent SQL injection.
- At the same time, executing same SQL query multiple times (with different parameters) increase the performance, because query parsing is done only once.

```
// prepare SQL statement
String sql = "SELECT * FROM books WHERE id=?"
PreparedStatement stmt = con.prepareStatement(sql);

// Execute SQL statement
stmt.setInt(1, id);
ResultSet rs = stmt.executeQuery();

// process resultset ...
```

CallableStatement

- CallableStatement is inheried from PreparedStatement.
- It is used to call stored procedures.
- These are also most commonly used to prevent SQL injection and improve performance of query execution.

MySQL stored procedure

- On MySQL prompt

```
DELIMITER $$

CREATE PROCEDURE SP_ADD_DEPT(D_NO INT, D_NAME VARCHAR(20), D_LOC VARCHAR(20))
BEGIN
      INSERT INTO DEPT VALUES(D_NO, D_NAME, D_LOC);
END;
```

```
$$

DELIMITER ;
```

- In Java application:

```
// prepare SQL statement
String sql = "{CALL SP_ADD_DEPT(?,?,?)}"
CallableStatement stmt = con.prepareCall(sql);

// Execute SQL statement
stmt.setInt(1, deptno);
stmt.setString(2, dname);
stmt.setString(3, loc);
boolean isResultSet = stmt.execute();

// process result ...
if(isResultSet) {
      ResultSet rs = stmt.getResultSet();
      // process resultset
} else {
      int cnt = stmt.getUpdateCount();
      // use update count
}
```

ResultSet

- While creating the statement, type and concurrency of result set can be specified.

ResultSet Types

- ResultSet.TYPE_FORWARD_ONLY
  - Only forward access, no reverse and random access.
  - This is default type.
- ResultSet.TYPE_SCROLL_INSENSITIVE
  - Can perform forward, reverse and random access.
  - Any changes done in the database while resultset is in use, will not reflect into resultset.
- ResultSet.TYPE_SCROLL_SENSITIVE
  - Can perform forward, reverse and random access.
  - Any changes done in the database while resultset is in use, will reflect into resultset.

ResultSet Concurrency

- ResultSet.CONCUR_READ_ONLY
  - default type
  - using result set records can be only fetched from database.
- ResultSet.CONCUR_UPDATEABLE
  - using resultset records can fetched and also manipulated (insert, update, delete)
  - operations can be performed on the current record).

ResultSet Functions

- ResultSet provide functions to fetch records and update records.

Fetch records

- boolean beforeFirst();
- boolean afterLast();
- boolean first();
- boolean last();
- boolean next();
- boolean previous();
- boolean absolute(int row);
- boolean relative(int relRow);

Check resultset position

- boolean isBeforeFirst();
- boolean isAfterFirst();
- boolean isFirst();
- boolean isLast();

Update functions

- deleteRow(): delete current row from resultset and database
- updateRow(): update changes made in current row into the database.
- rs.moveToInsertRow(): change resultset position to an empty row (which can be inserted later).
- insertRow(): add newly created row into the database.

CachedRowSet

- Inherited from ResultSet interface.
- Fetch all rows into the client memory, so that for accessing each row no need to connect to the database.
- In this case connection can be closed while data is accessed at the client side (after receiving CachedRowSet). Due to this more number of clients can connect to database.
- However if large of number of records are fetched into client memory, performance will be reduced (due to shortage of memory).

Transactions

- Transaction is set of SQL statements to be executed as a single unit.
- If all statements are executed successfully, then all of them should be committed to database. If any of the statement fails, then already executed statements should be rollbacked.
- RDBMS Transaction follow ACID properties.
    - Atomic
    - Consistency
    - Isolated
    - Durable
- RDBMS (Oracle/MySQL) have TCL (Transaction Control Language) to deal with transactions.
    - SAVEPOINT

- COMMIT
- ROLLBACK

- In JDBC, transactions are handled through methods of Connection class.
  - con.setAutoCommit()
  - con.commit()
  - con.rollback()
- Example

```
try {
      con.setAutoCommit(false);
      stmt1 = con.prepareStatement("UPDATE ...");
      stmt2 = con.prepareStatement("UPDATE ...")
      // ...
      stmt1.executeUpdate();
      stmt2.executeUpdate();
      con.commit();
} catch(Exception e) {
      con.rollback();
      // ...
} finally {
      // closing
}
```

Java EE (Enterprise Edition)
Web Programming

- Web Server:
  - Program that allows running multiple web applications in it.
  - Example: IIS, Apache, Jetty, Tomcat, ...
- Web Application:
  - Set of web pages which can be requested by the client & then served by the web server.
- Web Page:
  - Static contents: Hard-coded markup/tags -- HTML pages.
  - Dynamic contents: Contains a server side code, which upon client request executed on server side and produces response that is to be sent to the client.
- Web Client:
  - Consuming services given by web server.
  - Typical -- Web Browser:
    - request web page
    - render web page response
    - understand only HTML (+JS+CSS).

HTTP protocol

- HTTP is application layer protocol based on TCP protocol.
- HTTP is connection less protocol i.e. for each request from client a new socket connection is established for the server, through with request data is sent to the server. Server does process and produces response, which in turn sent back to the client through same socket and then connection is closed. For next request, new connection will be established.
- HTTP follows Request-Response model.

HTTP Request

- HTTP Request = Request headers + Request body
- Typical Headers
  - Request Method = GET/POST/PUT/DELETE/HEAD/TRACE/OPTIONS
  - Uri = resource identifier
  - HTTP version = 1.0 / 1.1
  - Browser language = en
  - Client/Browser = IP address
  - Content Type = type of data
  - Content Length = num of bytes in body
  - Cookies
  - User-Agent = Browser details
- Body contains
  - Data
    - e.g. form data in key=value pair.
    - e.g. JSON data (for REST services)

HTTP Response

- HTTP Response = Response headers + Resp body
- Typical Headers
  - HTTP version=1.0/1.1
  - Content-Type= type of data in resp body.
  - Content-Length= num of bytes in resp body.
  - Status code & message
  - Language
  - Server info= IP address
  - Cookies
- Body contains
  - data
    - e.g. HTML tags (for web page)
    - e.g. JSON data (for REST services)

HTTP Content Types

- text
  - text/plain, text/xml, "text/html"
- image
  - image/jpeg, image/png, ...
- audio
  - audio/mp3, audio/wav, ...
- video
  - video/mpeg, ...
- application
  - application/json, application/pdf, ...

HTTP Status codes

- 2xx: success e.g. 200 (success/OK)
- 3xx: redirection e.g. 302 (http redirection)
- 4xx: errors e.g. 403 (forbidden), 404 (not found), ...
- 5xx: server errors

HTTP Request Methods

- GET
  - To get a resource from server.
  - submit html form with method=get, click on hyperlink, accessing page from browser entering url.
  - If used with html form data, then data is sent to the server via url (query string -- url?key=value). Do not use this in html forms for sensitive data.
- POST
  - To submit a html form on server.
  - submit html form with method=post.
  - The data is sent in request body.
- PUT
  - To add/edit a file/resource on server.
- DELETE:
  - To delete a file/resource from server.
- HEAD
  - Server send only response headers (not response body).
  - Used to check resource availbility and info from server.
- TRACE
  - To debug.
- OPTIONS
  - To check which HTTP methods are supported by the resource.

Java Web Server

- Dynamic web pages are imeplemented in Java language and executed on the server side. The web server runs on JVM.
- HTTP request and HTTP response are accessed/represent as Java objects i.e. HttpServletRequest and HttpServletResponse.
- Java Web Server = Web Container + Extra Services
  - Web Container -- For execution of Servlet & JSPs.
  - Extra Services -- Connection pool, SSL (https), JNDI, ...
  - e.g. Tomcat, Lotus, ...
- Java Application Server = Web Container + EJB Container + Extra Services
  - Web Container -- For execution of Servlet & JSPs.
  - Extra Services -- Connection pool, SSL (https), JNDI, Transaction Managemnt, ...
  - EJB Container -- For execution of EJBs.
  - e.g. Glassfish, JBoss, WebLogic, WebSphere, ...

Apache Tomcat

- Tomcat is open-source Java web server by Apache foundation.
- Apache Tomcat directory structure
  - bin --> scripts/executables for start/stop tomcat.
  - lib --> tomcat jars -- impl of Java EE specs e.g. servlet-api.jar, ...
  - conf --> config --
    - server.xml: port, SSL, ...
    - tomcat-users.xml: security, ...
    - context.xml: webapps, ...
  - webapps --> Hot deployment directory
    - Copy/upload appln in this directory to deploy it on server.
  - logs --> Web server log.
  - temp --> temp dirs for execution
  - work --> temp dirs for execution -- JSP pages.

JavaEE Application

- According to JavaEE specification, the application should have fixed directory structure.
- Same directory structure should be packaged in .war file, while deploying application.

```
[webapp]
      |- *.html, *.jsp, ...
      |- *.jpg, *.mp3, ...
      |- *.js, *.css, ...
      |- [WEB-INF]
            |- web.xml (deployment descriptor)
            |- [classes]
            |     |- * *.class
            |- [lib]
            |     |- *.jar (third party)
            |- ...
```

Java Servlet

- Servlet is a Java class that is executed on the server side, when request comes from the client and produces response to be sent to the client.
- Servlet class must be inherited from javax.servlet.Servlet interface directly or indirectly.

interface javax.servlet.Servlet

- Servlet interface contains lifecycle methods.
- These methods should be implemented in servlet class and they are invoked by web-server.

Servlet life-cycle methods

- void init(ServletConfig config) throws ServletException;
  - Called only once (in life time of servlet object) after creation of the object (after constructor).
  - Should contain one-time init code e.g. Db connection, ...
- void service(ServletRequest req, ServletResponse resp) throws ServletException, IOException;
  - Called for each request.
  - Should contain response generation logic.
- void destroy();
  - Called only once (in life time of servlet object) before destroying the servlet object (before garbage collector is called).
  - Called when appln in undeployed or server is shutting down. Should contain de-init code e.g. Db dis-connection, ...

class GenericServlet

- Abstract class inherited from Servlet interface.
- Contains default impl of init() & destroy(); but service() is abstract.
- Protocol independent servlet.
- Two inherited classes: FtpServlet, HttpServlet.
- GenericServlet is also inherited from ServletConfig interface and Serialiable interface.

interface ServletConfig

- Represents servlet config given in web.xml file.

- Used to read that config in servlet code e.g. getInitParameter(), ...
- There is one ServletConfig object is created per servlet.

class HttpServlet

- Provides implementation of service() method as per HTTP protocol.
- This implementation internally calls appropriate doXyz() method of HttpServlet.
- Pseudo implementation of service() method in HttpServlet class is as follows.

```
void service(ServletRequest req, ServletResponse resp) throws ServletException,
IOException {
	// -- get http req data from header+body
	// -- create http resp: header+body
	this.service(req,resp);
	//
}

void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
	String method = req.getMethod();
	if(method == "GET")
		this.doGet(req, resp);
	else if(method == "POST")
		this.doPost(req, resp);
	else if(method == "PUT")
		this.doPut(req, resp);
	// ... so on
}

void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
	throw new NotImplementedException("...");
}
// doPost()
// doPut()
// ... so on
```

User-defined servlet class

- Typically user-defined servlet classes are inherited from javax.servlet.http.HttpServlet class.
- This class should override appropriate doXyz() methods. Most commonly doGet() and doPost() are implemented.

```
public class MyServlet extends HttpServlet {
	// ...
}
```

- This servlet class should also be configured in WEB-INF/web.xml as follows.

```
<web-app>
	<servlet>
		<servlet-name>First</servlet-name>
		<servlet-class>pkg.MyServlet</servlet-class>
	</servlet>
	<servlet-mapping>
```

```
            <servlet-name>First</servlet-name>
            <url-pattern>/hello</url-pattern>
        </servlet-mapping>
    </web-app>
```

Command-line Compilation/Execution

1. Create directory structure for web application (named as "first") in tomcat/webapps directory.
2. Create index.html in application directory.
3. Create MyServlet.java in WEB-INF/src directory.
4. Compile .java file to create .class file under WEB-INF/classes.
5. cd $TOMCAT_HOME/webapps/first/WEB-INF/src
6. export CLASSPATH=$TOMCAT_HOME/lib/servlet-api.jar
7. javac -d ../classes MyServlet.java

Eclipse project

- Add "Server Runtime" in Eclipse workspace.
  - Window --> Preferences --> Server --> Runtime Environments --> Add --> Select "Tomcat Installation Directory".
- Create New "Dynamic Web Project".
  - Give Project name.
  - Select Tomcat Runtime.
  - "Check mark" web.xml -- Deplomement descriptor.
  - Finish.
- In "Web Content" --> index.html
- In "src" --> MyServlet.java (within package)
- In "web.xml" --> add servlet config.
- Go to "index.html" --> right click --> Run As --> Run on Server.

Eclipse Maven project

- Maven web project should be created with same steps as mentioned in Maven section.

Servlet features
load-on-startup

- By default servlet obj is created when first request is made to that servlet.
- However using <load-on-startup> ensure that servlet object is created (and its init() is called) as soon as application is deployed.
- This is mainly useful for the servlets which contains some initialization for the whole application. Example: Spring DispatcherServlet create spring container, hence it is loaded/created as soon as application is deployed.

Servlet init parameters

- The configuration information can be associated with servlet as init parameters.
- It is useful for customizing settings for the servlet.
- The init parameters are declared in web.xml as follows.

```
<servlet>
```

```
        <servlet-name>First</servlet-name>
        <servlet-class>pkg.MyServlet</servlet-class>
        <init-param>
                <param-name>color</param-name>
                <param-value>cyan</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>First</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
```

- These parameters can be loaded into Java code using ServletConfig object getInitParameter() method.

```
String color = config.getInitParameter("color");
```

Servlet annotation configuration

- Servlets can also be configured using @WebServlet annotation instead of web.xml file.

```
@WebServlet(
        urlPatterns = "/hello",
        initParams = {
                @WebInitParam(name="color", value="pink")
        }
)
public class MyServlet extends HttpServlet {
        // ...
}
```

Request parameters

- When HTML form is submitted to servlet (using GET or POST method), its data is sent in HTTP request body.
- This data can be accessed using request object methods.

```
// For single values like textbox, radio-button, drop-down, ...
String value = req.getParameter("param-name");
// For multiple values like checkbox, multi-selection listbox, ...
String values = req.getParameterValues("param-name");
```

Inter-servlet navigation

- Control can be transferred from a servlet to another using three techniques.
  i. HTTP Redirection
  ii. RequestDispatcher: forward()
  iii. RequestDispatcher: include()

HTTP redirection

- Code

```
response.sendRedirect(url);
```

- When response class sendRedirect() is called, a temporary response (with status code 302 and destination url) is sent to the browser; due to which browser make a new request to the new link.
- In this case, two requests are originated from the browser and thus browser is aware of the navigation.
- This is slower process.

- It can redirect from any web component to any other web component of the same application or different application.

RequestDispatcher: forward()

- Code

```
// using ServletRequest
RequestDispatcher rd = request.getRequestDispatcher("url");
// OR using ServletContext
RequestDispatcher rd = servletContext.getRequestDispatcher("/url");
// then
rd.forward(req, resp);
```

- Only one request is originated from the client, and single response is given back.
- This is faster.
- Browser is not aware of the navigation.
- Navigation can be done only to the web components within the same application.
- The request object can be used to carry extra information using.
  - void setAttribute(String key, Object value);
  - Object getAttribute(String key);
- Request is forwarded to next servlet from which response will be given to the client.

RequestDispatcher: Include()

- This is similar to forward() approach, but use include() method instead of forward().
- Code

```
rd.include(req, resp);
```

- Request is sent to next servlet, which performs some processing and return back to the calling servlet.
- The response generated by the second servlet will be included into first servlet's response.

State Management

- HTTP is stateless protocol.
  - Since connection is closed after request, the information/state about the client is not maintained by the server.
- Maintaining information/state of the client across its multiple requests by the server is called as "State Management".
- The state management can be done on client side or server side.
- Client Side State Management objects
  - Cookies
  - Query String
  - Hidden Form Fields
  - HTML5 storage
- Server Side State Management objects
  - Session
  - Application/ServletContext
  - Request

Cookie

- Cookies store text data in key-value pair.

- Maximum cookie size is limited to 4 KB.
- Stored/accessed/manipulated from client.
- Used for storing non-sensitive information like user preferences.
- Once a cookie is sent from the server to the client, for each next request cookie will be sent back from the client to the server automatically (as per HTTP protocol).
- There are two types of cookies i.e. temporary and persistent.

Temporary cookie

- Stored in browser memory.
- When browser process is terminated, cookies are destroyed.
- This is default type.

Permanent cookie

- Stored in client disk.
- Need to set age of the cookie in seconds.
- Cookies are destroyed when time limit is expired.

Cookie coding

- Sending cookie

```
Cookie c = new Cookie("key", "value");
c.setMaxAge(3600); // for persistent cookie.
resp.addCookie(c);
Receiving cookie
Cookie[] arr = req.getCookies();
for(Cookie c:arr) {
      if(c.getName().equals("key")) {
            value = c.getValue();
            // ...
      }
}
```

- Removing cookie from client disk

```
c.setMaxAge(-1);
```

Session

- Session object (of HttpSession class) is used to store client state on server side.
- There is single session object for each client, that maintain data as key-value pairs.
- Session attribute name (key) is String, while value can be any Object.
- Session object is accessible using request.getSession() method.

Session coding

- Get session object
    - Session object is accessed using request.getSession() method.
    - This method check if cookie with name jsessionid is present. If present, get the session object of given id stored at server side (session map) and return it.
    - If cookie is not present, create a new session object and add into server session map with a new session id (unique 128-bit guid). Also create a cookie with name jsessionid and value as generated session id and send it to the client.

HttpSession session = req.getSession();

- To save value in session & retrieve back

```
session.setAttribute("key", value);
value = session.getAttribute("key");
```

- To destroy session object

```
session.invalidate();
```

Session Tracking

- Session tracking keeps track of which session object belong to which client.
- It is done using Cookie or URL rewriting.

Cookie based session tracking

- This is default session tracking mode.
- Internally create a cookie with name "jsessionid" & send session id to the client.
- This sessionid cookie is sent by client to the server with each request.

URL rewriting based session tracking

- This embeds sessionid into the url e.g. http://server:8080/app/page;jsessionid=823423?key=valuc
- It is enabled in web.xml

```
<session-config>
      <tracking-mode>URL</tracking-mode>
</session-config>
```

- Also URLs should be generated dynamically using response.encodeUrl() or response.encodeRedirectUrl().
- encodeUrl() add sessionid into the url and is useful for urls within application. e.g. <a href='url'>, <form action='url'> or request dispatcher.
- encodeRedirectUrl() add sessionid into the url only if url is of same application (not external). It is useful for response.sendRedirect() method.

```
modifiedUrl = resp.encodeUrl("url");
// OR
modifiedUrl = resp.encodeRedirectUrl("url");
```

Request

- For each client request a new HttpServletRequest object is created. It is destroyed, when response is given to the client. Note that request obj is alive while req is forwarded or included.
- So it can be used to transfer some data from current web component to the next web component where the request is to be forwarded/included.
- Like session, request also maintains attribute map i.e. <keyString, valueObject>.
- Note that request attributes are different than request parameters.

Request Parameter

- Comes from client in request body or url (query string).
- They are always string values.

```
value = req.getParameter("name");
values = req.getParameterValues("name");
```

Request Attribute

- Passed from one web component to another web component (serve side programming).
- They are of Object type.

```
req.setAttribute("key", value);
value = req.getAttribute("key");
```

ServletContext

- Represents the web application (/).
- One per web application.
- It can be accessed from most of the objects in the web application.

```
ctx = this.getServletContext(); // current servlet
ctx = config.getServletContext(); // servlet config
ctx = request.getServletContext();
ctx = session.getServletContext();
```

- The key-value pairs stored in servlet context can be accessed from any request to any web component of the application.

```
ctx.setAttribute("key", value);
value = ctx.getAttribute("key");
```

- Note that ServletContext can also be used for servlet navigation and accessing application level init parameters.

Servlet navigation

- It is used to create RequestDispatcher object.
- Note that URL path begin with context root " / ".

```
rd = ctx.getRequestDispatcher("/url");
rd.forward(req,resp);
//OR
rd.include(req,resp);
```

Application init/context parameters

- They represent configuartion for the whole web application.
- These values are declared in web.xml.

```
<context-param>
      <param-name>color</param-name>
      <param-value>pink</param-value>
</context-param>
```

- To access them in web application.

```
String color = ctx.getInitParameter("color");
```

Listeners

- Listeners are part of servlet specification.
- They Handle web application level events e.g. app start, app stop, session start, session stop, request begin, request end, ...
- There are variety of listener interfaces given to handle different events.

i. javax.servlet.ServletContextListener: Interface for receiving notification events about ServletContext lifecycle changes.
ii. javax.servlet.ServletContextAttributeListener: Interface for receiving notification events about ServletContext attribute changes.
iii. javax.servlet.ServletRequestListener: Interface for receiving notification events about requests coming into and going out of scope of a web application.
iv. javax.servlet.ServletRequestAttributeListener: Interface for receiving notification events about ServletRequest attribute changes.
v. javax.servlet.http.HttpSessionListener: Interface for receiving notification events about HttpSession lifecycle changes.
vi. javax.servlet.http.HttpSessionBindingListener: Causes an object to be notified when it is bound to or unbound from a session.
vii. javax.servlet.http.HttpSessionAttributeListener: Interface for receiving notification events about HttpSession attribute changes.

- Few example listener interfaces are as follows.

```
public interface ServletContextListener {
	void contextDestroyed(ServletContextEvent sce);
	void contextInitialized(ServletContextEvent sce);
}
public interface HttpSessionListener {
	void sessionCreated(HttpSessionEvent se);
	void sessionDestroyed(HttpSessionEvent se);
}
```

Listener coding

- Follow steps for implementing listener in web application.

i. Create a class inherited from listener interface(s).
   - Implement listener interface methods.
ii. Register the class with web application. This can be done using <listener> tag in web.xml or using @WebListener annotation on listener class.

- Example

```
public class MyListener implements ServletContextListener {
	// ...
}

<!-- web.xml -->
<listener>
	<listener-class>pkg.MyListener</listener-class>
</listener>
```

Filters

- Filters are also part of servlet specification.
- Filters is a way of implementing AOP (Aspect oriented programming) in Java EE.
- They have numerous applications including security, logging, performance measurement, etc.

Filter life-cycle

- Filter life cycle is similar to servlet life-cycle.
- Filter interface have three methods.
  - void init(FilterConfig filterConfig);
    - Called once in life-time of filter object.

- Contains initialization code.

- void doFilter(ServletRequest request, ServletResponse response, FilterChain chain);
  - Called for each request.
  - Typical implementation contain pre-processing, invoking servlet's service() method and then post-processing.
- void destroy();
  - Called once in life-time of filter object.
  - Contains de-initialization code.

Filter coding

- One web component can have one or more filters. Need multiple <filter> and <filter-mapping> for the same <url-pattern>.
- One filter can be used for multiple web components. Need multiple <filter-mapping> with same <filter-name> and different <url-pattern>.
- Alternatively filters can be configured using @WebFilter annotation.

```
public class MyFilter implements Filter {
	// ...
}

<!-- web.xml -->
<filter>
	<filter-name>My</filter-name>
	<filter-class>pkg.MyFilter</filter-class>
</filter>
<filter-mapping>
	<filter-name>My</filter-name>
	<url-pattern>/*</url-pattern>
</servlet-mapping>
```

Java Server Pages (JSP)

- JSP are out-dated. However they are still used in old applications.
- JSP are primarily used to implement presentation logic. But they can also contain business logic.
- JSP is front-end technology i.e. for Web based UI.
- Internally JSP is converted into the servlet during execution.
- Typically JSP page (.jsp) includes HTML tags, JSP tags, custom tags and Java code.
- However writing/embedding Java code in JSP pages is "worst" practice.
- Best practice is to use "Java Beans" with JSP pages for business logic.

JSP syntax

- JSP contains markup i.e. HTML tags, JSP tags, Custom tags.
- Server side Java code will be executed on server and produced reponse will be sent to the client.
- JSP have five server side syntaxes.
  i. Directive
  ii. Declaration
  iii. Scriptlet
  iv. Expression
  v. Comment

Directive

- Defines behaviour of JSP (generated servlet).
- There are three directives.
  - <%@ page ... %>
  - <%@ include ... %>
  - <%@ taglib ... %>

Declaration

- Java code (fields/methods) to be added in generated servlet.
- <%! ... %>

Scriptlet

- Java code to be executed per request i.e. to be added in service() method.
- <% ... %>

Expression

- Java expression (that evaluates to some value) whose result is embedded in generated response (writer).
- <%= ... %>

Comment

- Server side comment, which is discarded during processing and not visible to client.
- <%-- ... --%>

JSP Directives
@page directive

- language="java"
- contentType="text/html"
  - Will be converted to resp.setContentType("text/html");
- import="packagename"
- session="true"
  - true: Internally req.getSession() will be executed.
  - false: session object will be null.
- buffer="8kb" autoFlush="true"
  - All generated response will be stored in a temp buffer before response is sent to the client.
  - autoFlush=true: when response is generated or buffer is full, it will be automatically flushed to the client.
  - autoFlush=false: Exception will be thrown if buffer is full.
- isThreadSafe="true"
  - true: Single thread will be used service requests. Due to which thread synchronization problems may not occur.
  - false: For concurrent requests multiple threads will be created and multiple servlet objects will be created. This reduces performance. Internally generated servlet implement "SingleThreadModel" marker interface, which is deprecated with newer versions.
- isErrorPage="false"
  - false: simple jsp page.

- true: this page is used to display some error to the end user (custom error page). This page will have access to "exception" object.

- errorPage="myerr.jsp"
  - This attribute is used in normal jsp page (not in err page)
  - If current page get some error, web container will redirect to the given error page.

@include directive

- file="filename"
  - Include the file contents in current JSP. The file can be another JSP or HTML file.
  - This directive includes the file statically (unlike <jsp:include/>).

@taglib directive

- To use tags from the third party tag library.
- prefix="..."
  - Prefix to avoid clash between two tag libraries.
  - Any prefix other that standard prefix i.e. "jsp".
- uri="..."
  - Identifies tag library. As declared in .tld file of tag library.

JSP Life Cycle

1. Translation Stage
   - When first request is made for the JSP page, it will be loaded by the web container inside JSP engine.
   - JSP engine translate the JSP page into a servlet's java code. This .java file can be found in tomcat's "work" folder.
   - If there is any error in JSP syntax (e.g. scriptlets), then this stage fails.
2. Compilation Stage:
   - The translated servlet's .java file will be compiled into a .class file at runtime.
   - If there is any java syntax error, then this stage fails.
3. Instantiation (Loading) Stage:
   - The .class file will be loaded and object of the translated servlet will be created.
   - Immediately after this init method of the JSP i.e. jspInit() will be executed.
   - If this method throws any exception, this stage fails.
   - This stage is also called as Loading or Initialization stage.
4. Request Handling Stage:
   - All above stages are done only for the first request of the JSP file; However this stage is executed for each request.
   - For each request, _jspService() method is executed (which is made up of all the HTML tags, scriptlets and expressions in the JSP file).
5. Destruction Stage:
   - When servlet object is no longer used or web container is going down, jspDestroy() method will be executed after which servlet object will be garbage collected.

JSP Implicit objects

- Few objects are directly accessible in request handling stage (without need of their creation) - in scriptlets and expressions - called as "implicit objects".
- There are 9 implicit JSP objects:
  i. request: HttpServletRequest
  ii. response: HttpServletResponse
  iii. config: ServletConfig

iv. application: ServletContext
v. session: HttpSession
vi. out: JspWriter (Similar to PrintWriter)
vii. page: Object (Points to current JSP servlet object i.e. "this" pointer).
viii. pageContext: PageContext (To save some data on page scope)
ix. exception: Throwable (Accessible only in error pages)

JSP tags

- To minimize/eliminate Java code from JSP, it is recommended to use various "tags" instead of java code (scriptlet/expressions).
  i. JSP actions. <jsp:tagname ... />
  ii. Third party custom tags. <prefix:tagname ... />
  iii. User defined custom tag. <prefix:tagname ... />
- Typically tags wraps up Java code in markup code.

JSP Standard Actions

- The JSP having minimum scriptlets is considered as better JSP. So programmers should avoid writing scriptlets wherever possible.
- This can be done with the help of standard tags (actions), third party tags or custom tags.
- JSP built-in tags used for specific tasks. They have "jsp" prefix.

jsp:plugin and jsp:fallback

- These tags are used to embed applets into jsp pages.

```
<jsp:plugin type="applet" code="MyApplet.class" codebase="/appln" width="200"
height="300">
      <jsp:fallback>Applet Not Loaded</jsp:fallback>
</jsp:plugin>
```

- This code will be internally converted into html's <object> or <embed> tag.

jsp:foward, jsp:include and jsp:param

- jsp:include and jsp:forward internally calls requestDispatcher.include() and requestDispatcher.forward() respectively.

```
<jsp:forward page="pageurl"/>
<!-- OR -->
<jsp:include page="pageurl"/>
```

- jsp:include is used for dynamic inclusion of a file in JSP page.
- jsp:param is used to send additional request parameters to next page as follows.

```
<jsp:forward page="page2.jsp">
      <jsp:param name="key1" value="value1"/>
</jsp:forward>
```

- In next page page2.jsp, this request parameter can be accessed as follows.

```
String val = request.getParameter("key1");
```

jsp:useBean, jsp:setProperty, jsp:getProperty

- They are for using Java beans into JSP pages.

Java Bean

- Simple java classes holding business logic methods.

Implemening Java beans

- Write a simple java class having following members:
  i. One or more fields.
  ii. Parameter-less constructor.
  iii. Getter/setter methods (naming conventions must be followed).
  iv. One or more business logic methods.

JSP standard actions for Java beans

jsp:useBean

- jsp:useBean is for using java bean in JSP page.

```
<jsp:useBean id="obj" class="pkg.ClassName" scope="page"/>
```

- This tag check the whether Java bean object is present in the given scope with given name (id).
- If object is present, it will be accessed by that name.
- If object is not present, class will be loaded and object will be created at runtime.
- Then object will added into given scope with given name.

Java Bean scopes

1. application: Bean is stored into ServletContext and hence accessible in all requests to all pages by all users. This is highest scope.
2. session: Bean is stored into HttpSession and hence accessible in all requests to all pages by current user.
3. request: Bean is stored into HttpServletRequest and hence accessible in all the pages on which request is forwarded or included.
4. page: Bean is stored into PageContext and hence accessible in current request to current page. This is default and lowest scope.

jsp:setProperty

- Internally find corresponding setter method (i.e. setPropName()) and execute it on given bean object.

```
<jsp:setProperty name="obj" property="propName" value="fix_value"/>
<!-- OR -->
<jsp:setProperty name="obj" property="propName" param="req_param"/>
<!-- OR -->
<jsp:setProperty name="obj" property="*"/>
```

- "value" attribute is used to give fix value, while "param" attribute is used to give value from request parameter.
- If property is set to "*", all properties whose names are matching with request parameter names, are initialized with values of corresponding request parameter values.

jsp:getProperty

- Internally find corresponding getter method (i.e. getPropName()) and execute it. The return value will be added into html response.

```
<jsp:getProperty name="obj" property="propName"/>
```

- Before calling <jsp:setProperty> or <jsp:getProperty> the JSP page must have declared <jsp:useBean> tag.

Java beans example

- In CapBean.java

```
// java bean class
public class CapBean {
	// fields
	private String word;
	private String result;
	// param less ctor
	public CapBean() {
		this.word = "";
	}
	// getters/setters
	public void setWord(String word) {
		this.word = word;
	}
	public String getWord() {
		return this.word;
	}
	public void setResult(String result) {
		this.result = result;
	}
	public String getResult() {
		return this.result;
	}
	// business logic method
	public void convert() {
		result = word.toUpperCase();
	}
}
```

- In input.jsp

```
<form method="post" action="out.jsp">
	Word : <input type="text" name="in_word"/>
	<input type="submit" value="Submit"/>
</form>
```

- In output.jsp

```
<!-- create obj of bean -->
<jsp:useBean id="cb" class="pkg.CapBean" scope="page"/>
<!-- call setters to set property -->
<jsp:setProperty name="cb" property="word" param="in_word"/>

<!-- call getters to get property -->
Word : <jsp:getProperty name="cb" property="word"/>
<!-- call business logic method -->
<% cb.convert() %>
Result : <jsp:getProperty name="cb" property="result"/>
```

JSP Expression Language

- Used to access (read) values from different scopes without using scriptlet or expression syntax.
- EL syntax can also be used to evaluate arithmetic expressions, access fields/methods of scoped objects.
- Typical EL syntax is ${ ... }.

- All standard actions and EL syntax must be used outside scriptlets <% ... %>.
- All EL in the page can be ignored using <%@ page isELIgnored="true" %>.

EL syntax

- ${2 + 3 * 8 % 10}
- ${scopeName.objName}
  - Four EL scopes are: pageScope (lowest), requestScope, sessionScope, applicationScope (highest).
- ${objName}
  - In this case, the "obj" will be searched from the lowest scope to highest scope.
- ${scopeName.obj.propName}
  - To call getter methods of the object. e.g. ${sessionScope.lb.username}
- ${scopeName.objName.method()}
  - To call any method on the object.

EL Implicit objects

- EL can be used to access values from the special implicit variables (objects).
  i. ${param.varName}
     - request.getParamter("varName");
  ii. ${paramValues.varName}
     - request.getParamterValues("varName");
  iii. ${header.varName}
     - request.getHeader("varName");
  iv. ${headerValues.varName}
     - request.getHeaderValues("varName");
  v. ${initParam.varName}
     - application.getInitParameter("varName");
  vi. ${cookie.cookieName}
     - Will get the value of the cookie with cookieName
  vii. ${pageContext.request}, ${pageContext.session}, ...
     - To access objects like request, response, session, out from current page context.

JSTL

- Other than JSP standard actions, JSP specification also allows to implement JSP custom actions/tags.
- There are lot of custom tags are available as a part of different frameworks e.g. struts, spring, tiles, etc.
- JSTL is a standalone tag library given by Sun Microsystems.

JSTL components

- Core
  - Provide programming constructs and other commonly required tags like <c:if>, <c:forEach>, <c:out>, <c:url>, <c:redirect>, <c:set>, etc.
- Formatting
  - Provide date-time or number/currency formatting tags like <fmt:formatDate>, <fmt:parseDate>, <fmt:message>, <fmt:formatNumber>, etc.
- SQL
  - Provide SQL operations like <sql:setDataSource>, <sql:query>, <sql:update>, etc.
  - These tags are not commonly used, as they loose modularity approach.
- Functions

- Provide string functions like <fn:length()>, <fn:toUpperCase()>, <fn:substring()>, <fn:startsWith()>, etc.

- XML
  - Provide XML operations like <x:out>, <x:parse>, etc.

JSTL example

- step 1. Add jstl.jar into WEB-INF/lib directory OR make entry into Maven pom.xml

```
<dependency>
	<groupId>javax.servlet.jsp.jstl</groupId>
	<artifactId>jstl</artifactId>
	<version>1.2</version>
</dependency>
```

- step 2. At the start of the JSP page use "taglib" directive
  - In below example,
    - prefix = used to avoid tag name clashes
    - uri = identifier for tags collection (taglib).

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

- step 3. Use the appropriate tag with proper attributes into the JSP page. Few examples are as follows.

```
<!-- books.jsp -->
<c:forEach var="b" items="${bb.bookList}">
	<input type="checkbox" name="book" value="${b.bookid}"> ${b.name}
</c:forEach>
<!-- login.jsp -->
<c:choose>
	<c:when test="${lb.status=='true'}">
		//...
	</c:when>
	<c:otherwise>
		//...
	</c:otherwise>
</c:choose>
```

JSP Custom Tags

- Custom tags can be implemented to achieve functionalities which are not available into standard actions, JSTL or any other third party tab libraries.
- JSP has mainly two types of tags i.e. classic tags and simple tags.
- Both are inherited from the marker interface "JspTag".
- To implement these tags you can use interfaces "Tag" and "SimpleTag" or adapter classes "TagSupport" and "SimpleTagSupport" respectively.

Implementing SimpleTag

- step 1. Decide the use, name, attributes and body of the tag. Example: <wish> tag to print greeting message for the given name. e.g. <my:wish uname="some_name"/>

- step 2. Write the tag handler class inherited from "SimpleTagSupport" e.g. WishTag.

  i. Add param less constructor.
  ii. Add number of fields equal to number of attributes, with names matching to attribute.
  iii. Implement getter/setter for the fields.

iv. Override doTag() method of the "SimpleTagSupport" class and implement the business + presentation logic in it.

```
public class WishTag extends SimpleTagSupport {
	private String uname;
	public WishTag() {
		this.uname = "";
	}
	public void setUname(String uname) {
		this.uname = uname;
	}
	public String getUname() {
		return this.uname;
	}
	public void doTag() {
		JspContext ctx = this.getJspContext();
		JspWriter out = ctx.getOut();
		out.println("Hello, " + uname);
	}
}
```

- step 3. Write tag library descriptor (.tld) file inside WEB-INF to give complete information about the tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd">
<tlib-version>1.0</tlib-version>
<short-name>custom</short-name>
<uri>/WEB-INF/custtags.tld</uri>
<tag>
	<name>wish</name>
	<tag-class>sub.krd.bkshop.WishTag</tag-class>
	<body-content>empty</body-content>
	<attribute>
		<name>uname</name>
		<required>true</required>
		<rtexprvalue>true</rtexprvalue>
		<type>java.lang.String</type>
	</attribute>
</tag>
</taglib>
```

- step 4. Use the tag into the JSP page.

```
<%@taglib prefix="my" uri="/WEB-INF/custtags.tld" %>
// ...
<my:wish uname="${lb.username}"/>
```

SimpleTag life cycle

1. When page containing custom tag is accessed first time, during translation stage.
   - The tld file is referred (via given prefix & uri in @taglib)
   - From tld, used tag is found and validated (syntax).

2. For each time tag is used, tag-class (given in .tld file) is loaded and object is created at runtime. Its paramless constructor will be called here.
3. setJspContext() method will be called by the container and current JSP's PageContext object will be passed into it. This object contains all info needed to process JSP page.
4. If tag is child of any other tag, then its setParent() method will be called.
5. Then container calls setter methods for all attributes used in the JSP file during tag invocation.
6. If tag has some body, then setJspBody() method will be called to provide tag body.
7. Finally container calls doTag() method of the tag class, which does server side processing and generate html output if any.
8. After doTag() is completed, the tag's generated html will be added into page response.

- If not overridden, setJspContext() method of the "SimpleTagSupport" will be called (step 3), which will save the current JspContext; so that it can be accessed via call to getJspContext() [usually in doTag() method].

Web Programming/Design Models
Model I architecture

- Model I architecture is also called as "Model-View" Architecture.
- View contains Presentation Logic and typically implemented as JSP pages containing EL, JSTL & other tags.
- Model refers to the business logic components, known as Java Beans.
- Used for small web applications.

Model II architecture

- Also called as "Model-View-Controller" Architecture.
- View contains Presentation Logic and typically implemented as JSP pages containing EL, JSTL & other tags.
- Model refers to the Business Logic components, known as Java Beans.
- Controller is in-charge of navigation between the components. It is implemented as servlet or filters, which forwards request to the next JSP page.
- This architecture allows us to implement big but maintainable web applications.
- There are many MVC frameworks. Few popular one are Spring, JSF, etc.

Hibernate
Introduction

- Hibernate is an ORM tool.
- ORM: Object Relational Mapping:
  - Java Objects are mapped to RDBMS tables & their fields mapped to the table columns.
  - This automates SQL query generation, execution and result collection part.
  - Java has many ORM frameworks e.g. iBatis, Torque, Toplink, Hibernate, etc.
- Hibernate is most popular, open-source framework for ORM. It is hosted on sourceforge.net site.
- Internally hibernate works on multiple existing Java APIs including JDBC, JTA and JNDI services.

Architecture

- Important objects from the Hibernate layer are Configuration, SessionFactory, Session, Criteria, Query, Transaction.

Configuration

- The database configuration & other hibernate settings are typically stored in hibernate.cfg.xml file into CLASSPATH of the project.

- This file contains important parameters like connection.driver_class, connection.url, connection.username, connection.password, dialect, connection.pool_size, etc.
- Main configuration is JDBC settings (i.e. driver, url, username & password) and other settings are hibernate specific.
- Even though hibernate is abstracting details of underlying database, each database is different in terms of data types, SQL functions and minor query syntaxes. These differences need to be hidden from higher layers of hibernate. To accomplish this dialect class is used. The dialect class is responsible for actual query generation for specific database.
- All hibernate settings are read using Configuration object. In hibernate 4.0 & onwards, Configuration object is used with ServiceRegistry object.
- Main job of Configuration object is to create hibernate "SessionFactory".

SessionFactory

- SessionFactory object contains info required to create JDBC connection to the database.
- It is heavy-weight object and hence recommended to create single SessionFactory per database in a project. In case project is dealing with multiple databases, it will have multiple SessionFactory objects.
- To ensure single instance of SessionFactory, typical J2SE applications follow Singleton pattern to create SessionFactory object (e.g. HbUtil.java file).
- SessionFactory object is used to create hibernate Session in the application.

Session

- The Session object internally encapsulate JDBC connection.
- Since Session object is light-weight & represent live database connection, it is recommended to create Session object for each time database connectivity is needed. Also session should be closed immediately after current database operation is finished.
- This is most important object, as it provides functionality for all database operations.
- Database operations are done using various methods in Session object e.g. get(), load(), save(), persist(), update(), delete(), createQuery(), etc.

Criteria

- Used to fetch records from the database with some criteria.
- Criteria object internally modifies basic SELECT query with WHERE clause and/or ORDER BY clause.
- It is created using Session object (createCriteria() method).
- The ORDER BY clause is added using addOrder() method.
- The WHERE clause is added using add() method. It takes some Restrictions expression as an argument.
- For example, following code access all books of given subject in ascending order of id.

```
Criteria cr = session.createCriteria(Book.class);
cr.add(Restrictions.eq("subject", subject));
cr.addOrder(Order.asc("id"));
return cr.list();
```

DetachedCriteria

- The createCriteria() method is deprecated in Hibernate 4. However DetachedCriteria is still available.
- Its syntax is similar to Criteria.
- It is called as DetachedCriteria, because it is not associated with hibernate session. Thus it can be created outside DAO layer.

- Obviously this crieria cannot be executed directly. To execute, detachedCriteria.getExecutableCriteria(session) method is called to get Criteria object, which is executed as discussed earlier.

Query

- Hibernate supports its own query language called as HQL (Hibernate Query Language).
- This query language works on Entity objects rather than database tables.
- For example:
  - Created using session.createQuery("....");
  - "from Book b" --> Fetch all books from the BOOKS table.
  - "from Book b where b.subject=?" --> Fetch books of given subject from BOOKS table.
  - " update Employee e set e.age=? where e.name=?" --> Update age and name of employee record in database.
  - " delete from Account a where a.status=?" --> Delete all accounts of given status.

SQLQuery

- Hibernate also supports native SQL queries with this object.
- Created using session.createSQLQuery("....");

NamedQuery

- Commonly used HQL queries can be associated with entity objects using @NamedQuery parameter or in .hbm.xml file.
- Created using session.createQuery("....");
- These queries are converted into SQL, their syntax is validated and then reused for multiple times. It improves performance over simple HQL queries.

Transaction

- When multiple data manipulations (DML) are to be done as a single unit, transactions are used.
- Transaction ensures atomic execution of all queries, which are part of the transaction. If any of the query fails remaining queries are rollbacked.
- Typical hibernate code dealing with transactions is as follows:

```
Transaction tx = null;
try {
      tx = session.beginTransaction();
      // multiple DML operations like below like save(...) or update(...) or
delete(...);
      tx.commit();
} catch(Exception ex) {
      tx.rollback();
}
```

Hibernate Entity Life Cycle

- There are four states of any hibernate entity.
  - Transient
  - Persistent
  - Detached
  - Removed

Transient

- New entity object created in by java code (in application), are transient. In other words, hibernate session is not yet aware of these objects.

```
Book b = new Book(); // b is transient
```

Persistent

- Hibernate session maintains list of entity objects managed by it. It is called as "session cache".
- Transient/Detached objects can be added into session cache using certain functions. Also entity objects created by hibernate are by default part of session cache. All these objects are said to be "Persistent".
- Any changes into state of persistent objects are tracked by session. At the end of transaction, these changes are updated back in database automatically.
- Persistent objects cannot be garbage collected.
- If entity object requested by application is already present in session cache, it will be returned from there. Again query will not be fired on database.
- Transient objects can be made persistent using
  - session.save(), session.saveOrUpdate(), etc.
- Detached objects can be made persistent using
  - session.merge()

Detached

- Entity objects removed from session cache are said to be "Detached".
- They can be removed from cache using
  - session.close(), session.clear(), session.evict().

Removed

- Entity objects whose corresponding database row has been deleted are said to be "Removed".
- It is done using
  - session.delete()

CRUD operations

- Hibernate session provides various methods to perform database operations.

get() or load() method

- Both methods are used to search object in the table by primary key.
- get() method early fetch data/row from the database.
- get() method returns null, if object is not found.
- load() method returns proxy object instead of actually fetching data from database. The actual data will be fetched, when proxy object is accessed in the program.
- load() method throws exception, if object is not found.

save() or persist() method

- Both methods are used to insert a record in the database.
- save() method immediately insert record in database and returns primary key of the object. This method may be called from outside the transaction.
- persist() method insert record, but return type is void. This operation must be performed within transaction.

update() method

- Update record in database corresponding to the primary key.
- This operation must be performed within a transaction.
- It adds transient or detached into the session cache (and thus make it persistent).

delete() or remove() method

- Deletes corresponding record from the database.

merge() or saveOrUpdate() method

- If no object is found with given primary key, new object/record will be inserted in database; otherwise existing record is modified.
- Internally fires SELECT query on the database and based on result produced, it fires INSERT or UPDATE query.

Creating Hibernate SessionFactory

- SessionFactory code is also known as Bootstrapping code. Usually it is implemented as Singleton class.

Hibernate3 Bootstrapping

- Create Configuration object.
- Its configure() method read hibernate.cfg.xml and load its settings.
- Finally its buildSessionFactory() method creates a new SessionFactory.

```
public class HbUtil {
	private static SessionFactory factory = createSessionFactory();
	private static SessionFactory createSessionFactory() {
		Configuration cfg = new Configuration();
		cfg.configure();
		return cfg.buildSessionFactory();
	}
}
```

Hibernate5 Bootstrapping

- Newer Hibernate versions recommend creating SessionFactory using ServiceRegistry and Metadata objects.

```
public class HbUtil {
	private static final SessionFactory factory = createSessionFactory();
	private static ServiceRegistry serviceRegistry;

	private static SessionFactory createSessionFactory() {
		serviceRegistry = new StandardServiceRegistryBuilder()
				.configure() // read from hibernate.cfg.xml
				.build();
		Metadata metadata = new MetadataSources(serviceRegistry)
				.getMetadataBuilder()
				.build();
		return metadata.getSessionFactoryBuilder().build();
	}
}
```

ServiceRegistry

- It is resposible for configuring & managing hibernate services.
- The configuration can be done using one of the following ways.
  - Its configure() method reads hibernate.cfg.xml file (or can provide name of file).
  - Its loadProperies() method reads hibernate.properties file.
  - Its applySettings() method takes Map of configuration properties.
- There are few implementations of ServiceRegistry provided e.g. StandardServiceRegistry, BootstrapServiceRegistry, SessionFactoryServiceRegistry, etc.
- StandardServiceRegistry is most commonly used ServiceRegistry.

Metadata

- It is responsible for ORM mapping information.
- The configuration can be done using one of the following ways.
  - It can take settings from hibernate.cfg.xml through ServiceRegistry.
  - Its addResource() method can take path of .hbm.xml file.
  - Hibernate entity classes can be added using addAnnotatedClass() method.
  - Auto detectection of @Entity classes from packages is possible using addPackage() method.
- Finally it builds SessionFactory.

Creating Session

- Hibernate Session can be created using openSession() or getCurrentSession() method of SessionFactory class.

session = sessionFactory.openSession()

- Creates a new hibernate session.
- This session is associated with a JDBC connection.
- It can perform get() or load() method without needing a transaction.
- However DML operations can be performed only using transaction.
- Session must be closed at the end.

```
Transaction tx = null;
try {
      session = sessionFactory.openSession();
      tx = session.beginTransaction();
      // ... DML operations
      tx.commit();
} catch(Exception ex) {
      tx.rollback();
}
```

session = sessionFactory.getCurrentSession()

- Such Hibernate session is associated with some context (declared in hibernate.cfg.xml).
- Creates new session if called first time in given context. Subsequent calls to getCurrentSession() return the same session object.

- This session is not associated with JDBC connection. It will be associated with a JDBC connection when a new transaction is created. Hence any operation (CRUD) should be performed only within transaction. User should commit/rollback transaction, which in turn close/return JDBC connection.

```
try {
	session = sessionFactory.getCurrentSession();
	session.getTransaction().begin();
	// ... DML operations
	session.getTransaction().commit();
} catch(Exception ex) {
	session.getTransaction().rollback();
}
```

- Session is automatically closed at the end of context. Hence it should not be closed explicitly by the programmer.
- Possible session contexts are
  - thread
  - jta
  - managed

thread

- Session object is associated with thread TLS (Thread Local Storage).
- Session is automatically closed when thread is terminated.
- This is commonly used in standalone applications.

jta

- Session object is associated with external transaction.
- This is commonly used in JavaEE applications. JTA is supported by many application servers.

managed

- Session object is associated with custom transaction.

Hibernate Entity Relationships

- Typically RDBMS tables are related to each other via PK/FK constraints.
- There are four types of relationships.
  - One-to-one
  - One-To-Many
  - Many-To-One
  - Many-To-Many
- Hibernate fully support all of them with using annotations or .hbm.xml file.

OneToMany relation

- Example: One Dept has Many Employees.
- It is implemented by associating primary key of a table (parent) with foreign key of another table (child).

```
@Entity
public class Emp implements Serializable {
```

```
        // ...
        @Column(name="DEPTNO")
        private int deptno;
    }
    @Entity
    public class Dept implements Serializable {
        @Id
        private int deptno;
        // ...
        @OneToMany(mappedBy="deptno", fetch=FetchType.LAZY, cascade={CascadeType.ALL})
        private List<Emp> empList;
        // ... getters/setters
    }
```

FetchType

- The "fetch" attribute defines the select/fetch policy of child entities, when parent entity is fetched.
- It has two possible values i.e. LAZY and EAGER

LAZY

- When find/SELECT operation is done on parent table, query is fired to retrieve data of parent table only.
- When child list (e.g. empList) is accessed using getter method, another query is fired to retrieve data of child table.

EAGER

- When find/SELECT operaion is done on parent table, internally a JOIN query is fired to retrieve data of parent as well as child table.
- This works faster (as single database query is involved); all objects are fetched in memory at once and hence may consume more memory.

CascadeType

- The "cascade" attribute defines mainly manipulation policy of child entities, when parent entity is manipulated.
- It has five possible values i.e. PERSIST, REMOVE, MERGE, DETACH and REFRESH. It also allows a special value ALL.
- As names suggest these values represents transition between Hibernate entity states.
- They are associated with following Session methods.
    - PERSIST: session.persist()
    - REMOVE: session.remove()
    - MERGE: session.merge()
    - DETACH: session.evict()
    - REFRESH: session.refresh()

ManyToOne relation

- Example: Many Employees have Single Dept.
- Following example shows combination of OneToMany and ManyToOne relationship.

```
@Entity
public class Dept implements Serializable {
        @Id
```

```
            private int deptno;
            // ...
            @OneToMany(mappedBy="deptno", fetch=FetchType.LAZY,
cascade=CascadeType.ALL)
            private List<Emp> empList;
            // ... getters/setters
}
@Entity
@Table(...)
public class Emp implements Serializable {
            @Id
            private int empno;
            // ...
            @ManyToOne(fetch=FetchType.EAGER)
            @JoinColumn(name="DEPTNO") //EMPLOYEE.DEPTNO --> Database
            private Dept dept;
}
```

- As per above relationship, Emp object can be retrieved using.

```
Emp e = (Emp)session.get(Emp.class, emp_id);
// Will return Emp object & Dept obj within it - because of fetch type = EAGER.
```

- Also it is possible to add a new Dept object with multiple Emp objects in it.

```
Dept d = new Dept(0, "DEVELOPMENT", "PUNE"); //Ids is auto-generated
Emp e1 = new Emp(0, "ABC", 10000);
e1.setDept(d);
Emp e2 = new Emp(0, "PQR", 20000);
e2.setDept(d);
d.getEmpList().add(e1);
d.getEmpList().add(e2);
Transaction tx = session.beginTransaction();
session.persist(d);
tx.commit();
```

OneToOne relation

- Example: One Employee has One Address.
- Commonly it is implemented by having same primary key for both the tables.

```
class Address {
      // ...
      @OneToOne
      @PrimaryKeyJoinColumn
      private Emp e;
}

class Emp {
      // ...
      @OneToOne(mappedBy="e")
      private Address a;
}
```

ManyToMany relation

- Example: One Employee can have multiple meetings, while one meeting have multiple employees.

- This is typically implemented with a separate table having foreign keys of both tables.

```
class Emp {
	@Id @Column int empno;
	// ...
	@ManyToMany
	@JoinTable(name="EMPMEETINGS", joinColumns="EMPID", inverseJoinColumns="MEETID")
	List<Meeting> meetingList;
}

class Meeting {
	@Id @Column int meetid;
	// ...
	@ManyToMany(mappedBy="meetingList")
	List<Emp> empList;
}
```

Hibernate Inheritance

- There can be inheritance relation among entities. It can be implemented in four possible ways.
  - MappedSuperclass
  - SINGLE_TABLE strategy
  - TABLE_PER_CLASS strategy
  - JOINED strategy

@MappedSuperclass

- This annotation is helpful to mark super-class Hibernate entity.
- For this there will be different tables for sub-class entities.
- Note that super-class is not entity.
- In this implementation no polymorphic queries are possible. Also relations cannot be managed at super-class.
- step 1. Create database tables.
  - Create BOOK table with fields id, name, author, pages and price.
  - Create TAPE table with fields id, name, artist, duration and price.
- step 2. Implement entity classes. Note that super class is annotated using @MappedSuperclass

```
@MappedSuperclass
class Product {
	@Id
	@Column
	private int id;
	@Column
	private String name;
	@Column
	private double price;
	// ...
}

@Entity
@Table(name="BOOK")
class Book extends Product {
	@Column
```

```
	private String author;
	@Column
	private int pages;
	// ...
}

@Entity
@Table(name="TAPE")
class Tape extends Product {
	@Column
	private String artist;
	@Column
	private int duration;
	// ...
}
```

SINGLE_TABLE strategy

- This can be implemented when single table is maintained for all sub-class entities.
- In this, all classes including super class are entity classes.
- Note that relations can be created in super-class as well. Also polymorphic queries are allowed.
- step 1. Create database tables.
  - Create PRODUCT table with fields id, name, author, info, price and type.
- step 2. Implement entity classes.

```
@Entity
@Table(name="PRODUCT")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE")
class Product {
	@Id
	@Column
	private int id;
	@Column
	private String name;
	@Column
	private double price;
	// ...
}

@Entity
@DiscriminatorValue("BOOK")
class Book extends Product {
	@Column(name="author")
	private String author;
	@Column(name="info")
	private int pages;
	// ...
}

@Entity
@DiscriminatorValue("TAPE")
class Tape extends Product {
```

```
	@Column(name="author")
	private String artist;
	@Column(name="info")
	private int duration;
	// ...
}
```

TABLE_PER_CLASS strategy

- This is useful when different table for each sub-class & separate table for super-class is available. Usually super-class table is very small.
- All classes including super class are entity classes.
- Joins are fired internally to fetch data of sub-class entities.
- Note that relations can be created in super-class as well. Also polymorphic queries are allowed. However polymorphic queries will be slower due to joins.
- step 1. Create database tables as follows.
  - PUBLISHER: ID, NAME, ADDRESS
  - PRODUCT: PROD_ID, PUB_ID
  - BOOK: ID, NAME, AUTHOR, PRICE, PAGES
  - TAPE: ID, NAME, ARTIST, PRICE, DURATION
- step 2. Implement entity classes.

```
@Entity
@Table(name="PUBLISHER")
class Publisher {
	@Id
	@Column(name="id")
	private int id;
	@Column
	private String name;
	@Column
	private String address;
	@OneToMany(mappedBy="pubId")
	private List<Product> productList;
	// ...
}

@Entity
@Table(name="PRODUCT")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
class Product {
	@Id
	@Column(name="prod_id")
	private int id;
	@Column(name="pub_id")
	private int pubId;
	@Column
	private String name;
	@Column
	private double price;
	// ...
}
```

```
@Entity
@Table(name="BOOK")
class Book extends Product {
	@Column
	private String author;
	@Column
	private int pages;
	// ...
}

@Entity
@Table(name="TAPE")
class Tape extends Product {
	@Column
	private String artist;
	@Column
	private int duration;
	// ...
}
```

JOINED strategy

- This is useful when different table for each sub-class and separate table for super-class. Super-class table contains common data. Sub-class tables contains only specific data.
- All classes including super class are entity classes.
- Joins are fired internally to fetch data of sub-class entities.
- Note that relations can be created in super-class as well. Also polymorphic queries are allowed.
- step 1. Create database tables as follows.
  - PUBLISHER: ID, NAME, ADDRESS
  - PRODUCT: PROD_ID, PUB_ID, NAME, PRICE
  - BOOK: ID, AUTHOR, PAGES
  - TAPE: ID, ARTIST, DURATION
- step 2. Implement entity classes.

```
@Entity
@Table(name="PUBLISHER")
class Publisher {
	@Id
	@Column(name="id")
	private int id;
	@Column
	private String name;
	@Column
	private String address;
	@OneToMany(mappedBy="pubId")
	private List<Product> productList;
	// ...
}

@Entity
@Table(name="PRODUCT")
```

```
@Inheritance(strategy = InheritanceType.JOINED)
class Product {
	@Id
	@Column(name="prod_id")
	private int id;
	@Column(name="pub_id")
	private int pubId;
	@Column
	private String name;
	@Column
	private double price,
	// ...
}

@Entity
@Table(name="BOOK")
class Book extends Product {
	@Column
	private String author;
	@Column
	private int pages;
	// ...
}

@Entity
@Table(name="TAPE")
class Tape extends Product {
	@Column
	private String artist;
	@Column
	private int duration;
	// ...
}
```

Calling stored procedure

- Simple stored procedures (i.e. procedures with no OUT parameters) can be executed using native named queries of hibernate.
- The stored procedures with OUT parameters are called using JDBC code written in doWork() or doReturningWork() method in Hibernate 3.
- The stored procedures with OUT parameters are called using session.createStoredProcedureCall() in Hibernate 5.

Stored Procedures using NamedNativeQuery

- step 1. Implement stored procedure in database.

```
DELIMITER $$

CREATE PROCEDURE `SP_BooksByPriceGT`(p_price DOUBLE)
BEGIN
	SELECT ID,NAME,AUTHOR,SUBJECT,PRICE FROM BOOKS where PRICE > p_price;
END $$

DELIMITER ;
```

- step 2. In entity class use @NamedNativeQuery annotation. Note the @QueryHint org.hibenate.callable, which force Hibernate to call procedure using CallableStatement (instead of PreparedStatement).

```
@Entity
@Table(name="BOOKS")
@NamedNativeQuery(name="PricedBooks ",
query="{CALL SP_BooksByPriceGT (?)}", resultClass=Book.class,
hints={@QueryHint(name="org.hibernate.callable", value="true")})
class Book implements Serializable {
            // ...
}
```

- step 3. Access named query using session and execute it.

```
NamedQuery q = session.getNamedQuery("PricedBooks");
q.setParameter(0, 500.0);
List<Book> list = q.list(); // returns all books of price > 500
```

Stored Procedures with OUT parameter in Hibernate3

- step 1. Implement stored procedure in database.

```
DELIMITER $$

CREATE PROCEDURE `SP_GetBookPrice`(p_id INT, OUT p_price DOUBLE)
BEGIN
     SELECT PRICE INTO p_price FROM BOOKS where id = p_id;
END $$

DELIMITER ;
```

- step 2. Invoke stored procedure with JDBC syntax using doWork() or doReturningWork() method. Note that doWork() method takes functional interface Work whose execute() method returns void; while doReturningWork() method takes functional interface ReturningWork<T> whose execute() method returns generic type T.

```
Double price = session.doReturningWork(new ReturningWork<Double>() {
      public Double execute(Connection con) throws Exception {
            try(CallableStatement stmt = con.prepareCall("CALL SP_GetBookPrice(?,?)"))
{
                  stmt.setInt(1, id);
                  stmt.registerOutParameter(2, Types.DOUBLE);
                  stmt.execute();
                  return stmt.getDouble(2);
            }
      }
});
```

Stored Procedures with OUT parameter in Hibernate5

- step 1. Implement stored procedure in database.
- step 2. Register procedure parameters and execute procedure.

```
ProcedureCall sp = session.createStoredProcedureCall("SP_GetBookPrice");
sp.registerStoredProcedureParameter(0, Integer.class, ParameterMode.IN);
sp.registerStoredProcedureParameter(1, Double.class, ParameterMode.OUT);
sp.setParameter(0, id);
sp.execute();
```

```
return (String) sp.getOutputParameterValue(1);
```

Hibernate Query Language (HQL)

- Hibernate generates SQL queries based on constraints on fields and relations among entities. However certain times we need better control over generated SQL.
- HQL queries target Hibernate entities (not RDBMS tables). Syntactically they are similar to JPA-QL.
- HQL queries are represented by Query object and is declared on entity classes using @NamedQuery.

```
// ...
@NamedQuery(name="subjectBooks", query="from Book b where b.subejct = :p_subejct"),
public class Book {
      // ...
}
Query<Book> q = session.getNamedQuery("subjectBooks");
q.setParameter("p_subejct", subejct);
list = q.getResultList();
```

HQL SELECT queries

- Fetch all books
  - from Book b
- Fetch all books of given subject
  - from Book b where b.subject=:p_subject
- Fetch all books of given subject sorted by price in desc order
  - from Book b where b.subject=:p_subject order by b.price desc
- Select few columns of table
  - select b.id, b.name, b.price from Book b where b.subject=:p_subject
  - The selected columns are collected in Object[].
- Select few columns of table
  - select new Book(b.id, b.name, b.price) from Book b where b.subject=:p_subject
  - The selected columns are collected in Book object. The entity class should have constructor with appropriate arguments
- Executing analysis queries
  - select b.subject, sum(b.price) from Book b group by b.subject
  - The selected columns are collected in Object[].

HQL DML queries

- Delete books of given subject
  - delete from Book b where b.subject=:p_subject
  - This query is executed using query.executeUpdate() object.
- Increase price of all books by 5 percent.
  - update Book b set b.price=b.price+b.price*0.05
  - This query is executed using query.executeUpdate() object.
- Insert into table from another table.
  - insert into OldBook ob select from Book b
  - This query is executed using query.executeUpdate() object.
  - Note that insert query with arbitrary columns is not supported in HQL.

Auto-generated Id (Primary Key)

- Different RDBMS have differnent methods for generating Primary keys while inserting new record. E.g. AUTO_INCREMENT, IDENTITY, SEQUENCE, UUID, HILO, etc.

- Hibernate associate such auto-generated ids to @Id field using JPA compliant @GeneratedValue annotation.
- There are four main strategies for auto-generating ids.
  - AUTO
  - IDENTITY
  - TABLE
  - SEQUENCE

AUTO

- Id generation strategy is chosen automatically by JPA provider (dialect).
- Hibernate 5.3 with MySQL5Dialect use RDBMS table for auto ids.
- step 1. Create generator table into MySQL.

```
CREATE TABLE gen(next_val INT PRIMARY KEY);
INSERT INTO gen VALUES(1000); -- Initial value.
```

- step 2. In entity class apply @GeneratedValue on @Id column.

```
@GeneratedValue(generator="genName", strategy=GenerationType.AUTO)
@Id
private int id; // column-field
```

- This internally selects "next_val" as id value (with pessimistic locking to avoid duplicated ids for concurrent transactions) and update that id (+1).

IDENTITY

- This is mapped to MySQL AUTO_INCREMENT column.
- step 1. Create table with AUTO_INCREMENT column.

```
CREATE TABLE MyEntity(id INT PRIMARY KEY AUTO_INCREMENT, ...);
```

- step 2. In entity class apply @GeneratedValue on @Id column.

```
@GeneratedValue(generator="gen", strategy=GenerationType.IDENTITY)
@Id
private int id;
```

- Generated INSERT query doesn't add Primary key column. It is auto-generated by database.

TABLE

- Use dedicated RDBMS table to generate Primary keys for multiple tables/entities.
- Its working is similar to AUTO strategy (for MySQL).
- step 1. Create table for auto-generated ids.

```
CREATE TABLE id_gen(name VARCHAR(20) PRIMARY KEY, id INT);
```

- step 2. In entity class apply @GeneratedValue on @Id column.

```
@TableGenerator(name="gen", initialValue=100, pkColumnName="name",
valueColumnName="id", allocationSize=1, table="id_gen")
@GeneratedValue(generator="gen", strategy=GenerationType.TABLE)
@Id
private int id;
```

SEQUENCE

- Used with RDBMS like Oracle which support SEQUENCE.
- step 1. Create SEQUENCE for auto-generated ids.

```
CREATE SEQUENCE myseq START WITH 1 INCREMENT BY 1 CACHE 1;
```

- step 2. In entity class apply @GeneratedValue on @Id column.

```
@SequenceGenerator(name="gen", sequenceName="myseq", allocationSize=1)
@GeneratedValue(generator="gen", strategy=GenerationType.SEQUENCE)
@Id
private int id;
```

Auto-generated Id (Hibernate specific)

- Apart from JPA compliant @GeneratedValue, hibernate also provides @GenericGenerator with few more possible strategies to generate ids.
- Its general syntax is as follows.

```
@GenericGenerator(name="g", strategy="...")
@GeneratedValue(generator="g")
private int id;
```

- The possible strategies are
  - native: like AUTO (depends on database dialect).
  - guid: auto generated unique GUID/UUID (128-bit id).
  - hilo: id generated using high low algorithm.
  - identity: mapped to AUTO_INCREMENT or IDENTITY column of database.
  - sequence: id generated using sequence.
  - increment: generate id as MAX(id)+1.
  - foreign: id corresponding to foreign key.

Composite Primary Key

- The RDBMS table Composite Primary key are declared as @Embeddable class and declared as @EmbeddedId in entity class.
- Note that @Embeddable class must be Serializable.
- step 1. Create RDBMS table with Primary key.

```
CREATE TABLE STUDENTS (roll int, std int, name varchar(50), marks double, primary
key(roll, std));
```

- step 2. Create Composite Primary key class.

```
@Embeddable
public class StudStdRoll implements Serializable {
	private static final long serialVersionUID = 1L;
	@Column
	private int std;
	@Column
	private int roll;
	// ...
}
```

- step 3. Create entity class.

```
@Entity
@Table(name="STUDENTS")
public class Student implements Serializable {
      private static final long serialVersionUID = 1L;
      @EmbeddedId
      private StudStdRoll stdRoll;
      @Column
      private String name;
      @Column
      private double marks;
}
```

Session Factory Cache

- Session object is associated with a cache, also called as first level cache. Each hibernate request pass through session cache.
- Each hibernate session factory can also be associated with cache of recently used objects. It is also called as second level cache.
- Since there is single session factory in whole application, objects present in this cache can be accessed from any other session object. This will further improves speed of execution.
- However SessionFactory cache is optional. Hibernate session factory cache can be configured in hibernate.cfg.xml file as "hibernate.cache.provider_class".
- Important second level cache types are: EHCache, OSCache, SwarmCache and JBoss Cache.
- Cache strategies are: ReadOnly, ReadWrite, NonstrictReadWrite, Transactional.

Cache Strategy

- READ_ONLY: Used only for entities that never change (exception is thrown if an attempt to update such an entity is made). It is very simple and performant. Very suitable for some static reference data that don't change.
- NONSTRICT_READ_WRITE: Cache is updated after a transaction that changed the affected data has been committed. Thus, strong consistency is not guaranteed and there is a small time window in which stale data may be obtained from cache. This kind of strategy is suitable for use cases that can tolerate eventual consistency.
- READ_WRITE: This strategy guarantees strong consistency which it achieves by using 'soft' locks: When a cached entity is updated, a soft lock is stored in the cache for that entity as well, which is released after the transaction is committed. All concurrent transactions that access soft-locked entries will fetch the corresponding data directly from database
- TRANSACTIONAL: Cache changes are done in distributed XA transactions. A change in a cached entity is either committed or rolled back in both database and cache in the same XA transaction.

Cache types

- EHCache: It can cache in memory or on disk and clustered caching and it supports the optional Hibernate query result cache.
- Swarm Cache: A cluster cache based on JGroups. It uses clustered invalidation, but doesn't support the Hibernate query cache.
- OSCache (Open Symphony Cache): Supports caching to memory and disk in a single JVM with a rich set of expiration policies and query cache support.
- JBoss Tree Cache: A fully transactional replicated clustered cache also based on the JGroups multicast library. It supports replication or invalidation, synchronous or asynchronous communication, and optimistic and pessimistic locking.

- Note that not all cache support all strategies.

Implementation	Read-only	Nonstrict-read-write	Read-write	Transactional
EH Cache	Yes	Yes	Yes	No
OS Cache	Yes	Yes	Yes	No
Swarm Cache	Yes	Yes	No	No
JBoss Cache	Yes	No	No	Yes

EHCache Usage

- step 1. Add cache impl jars in classpath -- pom.xml (ehcache)

```
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-ehcache</artifactId>
	<version>5.3.10.Final</version>
</dependency>
```

- step 2. In hibernate.cfg.xml enable & configure ehcache.

```
<property
name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFac
tory</property>
<property name="hibernate.cache.use_second_level_cache">true</property>
```

- step 3. Enable caching for entity class objects.

```
@Cacheable
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY)
@Entity
class Book {
	// ...
}
```

Query Cache

- Typically used with second level cache.
- Need to enable in hibernate.cfg.xml.

```
<property name="hibernate.cache.use_query_cache">true</property>
```

- Stores HQL query results. Only ids are stored corresponding to the query. The entities can be cached in SessionFactory cache.
- Caching of queries is done programmatically.

```
q.setCacheable(true);
```

Forward and Reverse Engineering
Reverse Engineering

- It is also called as "Table first" approach.
- From database tables generate hibernate entity classes and relations.
- This is most common approach, because software development usually begin with database design.
- This can be done in two ways.

- Using annotations or mapping file manually.
- Using automated JPA tools like JBoss Hibernate tools, which automatically generate hibernate entity classes along with proper relationships, as per PK/FK constraints on database tables.

Forward Engineering

- It is also called as "POJO first" approach.
- From hibernate entities generate database tables.
- For this, first implement hibernate entity classes with all required relationships. Hibernate can generate database schema by configuring "hibernate.hbm2ddl.auto" in hibernate.cfg.xml.

hibernate.hbm2ddl.auto

- create: Database dropping will be generated followed by database creation.
- create-only: Database creation will be generated.
- create-drop: Drop the schema and recreate it on SessionFactory startup. Additionally, drop the schema on SessionFactory shutdown.
- update: Update the database schema.
- validate: Validate the database schema
- none: No action will be performed.

Hibernate and JPA

- JPA (Java Persistence API) is specification for Java ORM frameworks.
- Hibernate is most popular JPA provider.
- JPA concepts are mapped to Hibernate concepts.

persistence.xml

- The persistence.xml contains configuration of database and other ORM configuration.
- It contains database connection properties along with provider implementation and may contain provider related configuration.

```
<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
```

EntityManagerFactory

- EntityManagerFactory is built from settings in persistence.xml.
- It is similar to Hibernate SessionFactory.

EntityManager

- EntityManagerFactory creates EntityManager.
- It is similar to Hibernate Session object.
- Objects in EntityManager are managed by JPA i.e. their changes are automatically updated in database.
- EntityManager have several methods e.g. find(), persist(), merge(), remove(), createQuery(), etc.

JPA-QL

- JPA QL is query language of JPA, which is similar to Hibernate QL (HQL).

JPA entity life cycle

- JPA entity life cycle is similar to Hibernate entity life cycle.
- Entities may have four states.
  - New (Same as Hibernate Transient)
  - Managed (Same as Hibernate Persistent)
  - Detached (Same as Hibernate Detached)
  - Removed (Same as Hibernate Removed)

Criteria Queries

- Hibernate deprecated Criteria, but came up with Criteria queries.
- The Criteria was limited to SELECT operation, while Criteria queries can also be used with DML operations.
- The Criteria queries are JPA compliant.
- There are three main Criteria queries.
  - CriteriaQuery<>
  - CriteriaUpdate<>
  - CriteriaDelete<>

CriteriaQuery<>

- It is used for SELECT operation.
- Conceptually it is similar to older Criteria objects.
- It follows builder design pattern and eliminates need of writing query (neither SQL nor HQL/JPAQL).

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Book> crQuery = builder.createQuery(Book.class);
Root<Book> table = crQuery.from(Book.class);
crQuery
.select(table)
.where(builder.equal(table.get("author"), author));
list = em.createQuery(crQuery).getResultList();
```

CriteriaUpdate<>

- It represents criteria Query for UPDATE operation.
- It follows builder design pattern and eliminates need of writing query (neither SQL nor HQL/JPAQL).

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaUpdate<Book> crUpdate = builder.createCriteriaUpdate(Book.class);
Root<Book> table = crUpdate.from(Book.class);
crUpdate.set(table.<Number>get("price"),
            builder.sum(table.<Number>get("price"),
builder.quot(table.<Number>get("price"), 5)))
 .where(builder.equal(table.get("subject"), subject));
 cnt = em.createQuery(crUpdate).executeUpdate();
```

CriteriaDelete<>

- It represents criteria Query for DELETE operation.
- It follows builder design pattern and eliminates need of writing query (neither SQL nor HQL/JPAQL).

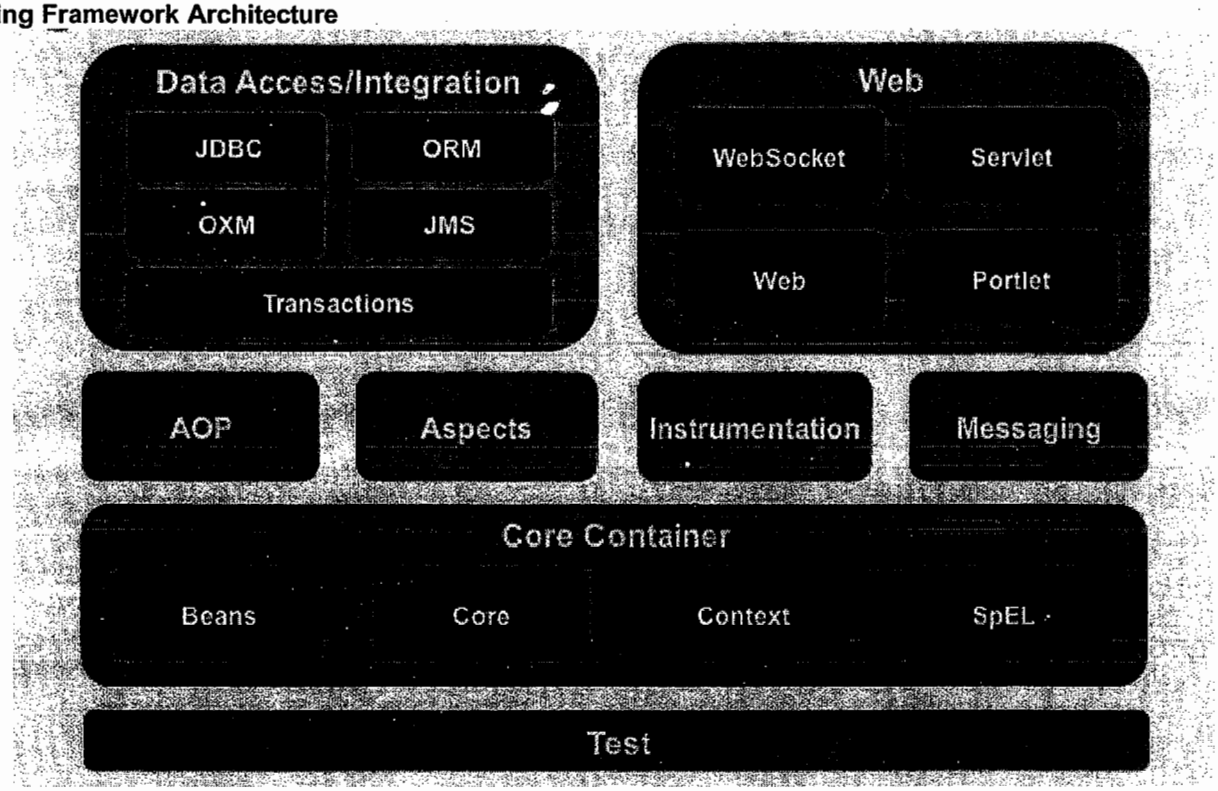
```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaDelete<Book> crDelete = builder.createCriteriaDelete(Book.class);
Root<Book> table = crDelete.from(Book.class);
```

```
crDelete.where(builder.equal(table.get("subject"), subject));
cnt = em.createQuery(crDelete).executeUpdate();
```

Spring Framework

- Spring is lightweight comprehensive framework for simplifying JavaSE and JavaEE application development.

Spring Framework Architecture



Spring Framework Features

- Modular: Spring framework has multiple Jar files. As per need of application developer can choose appropriate jars.
- Flexible: Most of frameworks follow "all or nothing" policy. Spring framework modules can be used well with any of the existing framework.
- Well Designed: Using this Framework highly extendible applications can be developed.
- Design with Interfaces: Interfaces are immutable. Spring recommends designing interface and then spring beans. This ensures that implementations can be changed easily.
- Simplified: Most of spring beans are simple POJO classes. Hence developing application is far easier.
- Declarative AOP: Aspect oriented programming is supported using annotations also.
- Configuration: Provides differnent methods of configuration i.e. XML, annotations or mixed configuration.
- Maintainable: Applications are easier to test. Built in support provided for unit testing. Also all exceptions are unchecked and highly readable.
- Dependency Injection: The spring core provide facility of dependency injection, which is heart of spring framework.

Dependency Injection

- Dependency injection feature that allow to initialize fields automatically while object creation. The fields (i.e. inner objects) are called as dependencies, while outer object is dependent object.
- Typical java applications create an object, initialize its fields (using constructor or setters) and then invoke business logic methods on that object.
- Since dependency injection feature is inverse as of typical applications initializing the objects, this is also considered as "Inversion of Control". Dependency injection is main feature of spring container, so spring container is also referred as "IoC container".
- This feature handles initialization of the object and allows programmers to focus on business logic code.
- There are three types of dependency injection
  - Setter Based dependency injection
  - Constructor Based dependency injection
  - Field Based dependency injection

Setter Based dependency injection

- Dependencies are initialized using setter methods.
- Spring bean configuration xml file example.

```
<bean id="b1" class="pkg.Box">
      <property name="length" value="5"/>
      <property name="breadth" value="4"/>
      <property name="height" value="3"/>
</bean>
```

- When bean object is created by spring container, it invokes object's setLength(), setBreadth() and setHeight() methods and initialize its fields.
- To initialize one bean object (instead of fixed value) as dependency of another object, use "ref" attribute of property element.
- For example, if p1 is already created bean (of Person class) in spring bean configuration file, then it can be used to as dependency of another bean object (in Account class).

```
<bean id="a1" class="pkg.Account">
      <property name="accId" value="101"/>
      <property name="balance" value="10000.00"/>
      <property name="type" value="Saving"/>
      <property name="accHolder" ref="p1"/>
</bean>
```

Constructor Based dependency injection

- Dependencies can also be initialized using parameterized constructor.
- Spring bean configuration xml file example.

```
<bean id="b2" class="pkg.Box">
      <constructor-arg index="0" value="1"/>
      <constructor-arg index="1" value="2"/>
      <constructor-arg index="2" value="3"/>
</bean>
```

- When bean object is created by spring container, it invokes parameterized constructor and initialize its fields.

Field Based dependency injection

- Dependencies can also be assigned to field using @Autowired or @Value.
- This is discussed later.

Spring ApplicationContext

- Before Spring 3.x ean objects were created using bean factory (i.e. XMLBeanFactory). Due to its limitations, XMLBeanFactory is deprecated in Spring 4.x.
- Spring bean objects are now created using application context.
- When context is loaded, all singleton bean objects are created.
- Calling getBean() method on application context returns bean object reference.
- Shutdown hook can be registered with JVM, so that at the end of application spring container can destroy all singleton bean objects automatically.
- Support for event mechanism as well as internationalization is available with application context.
- Some important application contexts are shown below.

```
ApplicationContext
      |- ClassPathXmlApplicationContext
      |- FileSystemXmlApplicationContext
      |- AnnotationConfigApplicationContext
      |- WebApplicationContext
                  |- XmlWebApplicationContext
                  |- AnnotationConfigWebApplicationContext
```

- Here are details of few types of application contexts.
  - ClassPathXmlApplicationContext:
    - Bean objects are declared in spring bean configuration file.
  - AnnotationApplicationContext:
    - Bean objects are declared in a separate configuration class (annotated with @Configuration class).
  - WebApplicationContext:
    - Used in spring web mvc applications.

Spring Bean Scopes

- Spring bean scope can be configured in bean configuration file or using @Scope annotation.
- There are four scopes i.e. singleton, prototype, request and session.
- singleton:
  - Singleton is default scope of bean object.
  - When application context is loaded, all singleton bean objects are created.
  - Call to getBean() will always return same object's reference (no new object created).
- prototype:
  - When application context is loaded, bean object is not created.
  - Each call to getBean() will create a new object and return it.
- request:
  - Used only in web mvc applications.
  - Scope of the bean is limited to current request.
- session:
  - Used only in web mvc applications.
  - Scope of the bean is limited to current user session.

Bean Configuration Annotations

- From Spring 3.0 onwards, beans can also be configured using annotations.
- These annotations are activated if spring bean configuration file contains.

```
<context:annotation-config/>
```

- @PostConstruct
  - Used on method in bean class. This method is called by spring container after object creation and dependency injection.
- @PreDestroy
  - Used on method in bean class. This method is called by spring container while destroying bean object.
- @Autowired
  - Bean dependency objects (fields) can be annotated using @Autowired.
  - In this case, matching bean object is attached (dependency injection) with dependent object.
  - Matching bean means bean of the same type or derived type.
  - In case of no matching bean found or multiple beans found, exception is raised.
  - Note that @Autowired can be applied on field, setter or constructor (single argument) level.
- @Qualifier("beanName")
  - Can be used with @Autowired.
  - In case of multiple matching beans found for autowiring, exception is raised.
  - It can be resolved using @Qualifier and specifying name of bean to be attached/injected.

Spring Configuration

- Before Spring 3.x bean configuration was limited to XML files. From Spring 3.x onwards annotations are also used for the configuration.
- There are three types of configuration
  - XML based configuration
  - Annotation based configuration
  - Java based configuration

XML based configuration

- Spring bean configuration file is created in classpath or filesystem (as discussed earlier).
- ClassPathXmlApplicationContext or FileSystemXmlApplicationContext used to read the file and create beans accordingly.

Annotation based configuration

- Beans are configured using annotations like @Required, @PostConstruct, @PreDestroy, @Autowired, @Qualifier (as discussed later).
- Usually annotations are used along with XML bean configuration file. In this case, it is also considered as mixed configuration.

Java based configuration

- No XML file is used for bean configuration.
- The bean creation is done with special configuration class annotated with @Configuration. The class contains methods to create beans annotated with @Bean.
- This is most popular method of configuration nowadays.

```
@Configuration
public class AppConfig {
```

```
	@Bean
	public BoxImpl b1() {
		BoxImpl b = new BoxImpl();
		b.setLength(10);
		b.setBreadth(8);
		b.setHeight(6);
		return b;
	}
	@Bean
	public BoxImpl b2() {
		BoxImpl b = new BoxImpl(5, 4, 3);
		return b;
	}
}
```

- In order to auto-detect stereo-type annotations, the configuration class should be annotate with @ComponentScan.

```
@Configuration
@ComponentScan(basePackages = { "com.sunbeaminfo.sh" })
public class MyAppConfig {
	// ...
}
```

Auto-wiring

- Auto-wiring is automatically injecting appropriate dependency beans into the dependent beans.
- Auto-wiring can be done in XML based configuration as well as annotation/Java based configuartion.

Auto-wiring with XML based configuartion

- Typical syntax in bean configuartion file (of using autowire attribute) is as follows.

```
<bean id="b" class="..." autowire="default|no|byType|byName|constructor"/>
```

- Possible autowire values are as follows.
  - "default" or "no": Autowiring is disabled.
  - "byType": Dependency bean of property type will be assigned (via setter).
    - If multiple beans of required type are available, then exception is thrown.
    - If no bean of required type is available, autowiring is not done.
  - "byName": Dependency bean of property name will be assigned (via setter).
    - if no bean of required name is available, autowiring is not done.
  - "constructor": Dependency bean of property type will be assigned via single argument constructor (of bean type).
- There is also autowire-candidate attribute for bean configuration.
  - If it is false, the dependency bean is not considered for auto-wiring.
  - Its default value is true.

```
<bean id="b" class="..." autowire-candidate="true|false" .../>
```

Auto-wiring with Annotation based configuartion

- As discussed earlier.

Stereo Type Annotations

- It is not compulsory to declare each spring bean in bean configuration file.
- To automatically detect spring beans in certain package (and its sub-packages) of application, stereo-type annotations are used.
- These annotations are activated if spring bean configuration file contains:

```
<context:component-scan base-package="packagename"/>
```

- There are four stereo-type annotations used for different represent separate layers in any application.
- @Component:
  - Used for simple POJO classes used in application.
  - They can be models, entities or any simple java classes.
- @Repository:
  - Used for data access layer in application.
  - Typically they are JDBC Dao, Hibernate Dao or simply Collection/File handling classes.
- @Service:
  - Contains core business logic of the application.
  - Typically they access repository classes inside them.
- @Controller:
  - Used for web mvc applications.
  - Contains navigation logic and model-view interaction.
  - Typically use service classes inside them.

Spring Expression Language (SPEL)

- SPEL can be used in spring bean classes using @Value annotation.
- Syntax:
  - #{var.member} --> invokes getter method on object var.
  - #{var.method()} --> invokes method() on object var.
- In bean classes @Value annotation can be used for dependency injection (instead of @Autowired and @Qualifier).

```
//BookDao.java
@Repository("bkDao")
public class BookDao {
	// ...
}
//BookService.java
@Service
public class BookService {
	@Value("#{bkDao}")
	private BookDao dao;
	// ...
}
```

- However, using @Value instead of @Autowired reduces performance. So use @Value attribute only when @Autowired cannot be used.

```
//Person.java
@Component("p")
public class Person {
```

```
        private String name;
        private String address;
        // ...
}
//AadharCard.java
@Component("a")
public class AadharCard {
        private String aadharId;
        @Value("#{p.name}")
        private String name;
        // ...
}
```

Spring JUnit integration

- Unit Testing is testing each functioality of the project/module individually.
- It is part of development process. Test Driven Development (TDD) is commonly used approach.
- In JUnit 4.x, test classes are written using @RunWith annotation, while each test case method is annotated with @Test annotation.
- Other than @Test, few more annotations can be used on methods as follows.
    - @Before: This method executes before each test case.
    - @After: This method executes after each test case.
    - @BeforeClass: This method executes once before all test cases (for one time setup).
    - @AfterClass: This method executes once after all test cases.
- When test case fails, AssertionError is thrown. This assertion is typically done using static methods of org.junit.Assert class e.g. assertTrue(), assertFalse(), assertNotNull(), fail(), etc.

Spring JUnit integration steps

- step 1. Add spring-context, spring-test and junit dependencies in pom.xml.
- step 2. Implement test class and test cases for desired logic.

```
 @RunWith(SpringJUnit4ClassRunner.class)
 @ContextConfiguration(loader=GenericXmlContextLoader.class
locations="classpath:beans.xml") // creates spring context while test
 public class MyTest {
        // autowire desired objects here
        @Test
        public void testAddCustomer() {
                // test case code
                assertFalse(condition, "test case failed");
        }
}
```

Spring AOP

- In aspect oriented programming, the business logic is separated from cross-cutting concerns. The cross-cutting concerns are different aspects other than core business logic e.g. logging, monitoring, security, etc.
- Typical cross-cutting concerns are implemented before (pre-processing) or after (post-processing) execution of core business logic.
- Due to this, code will be modularized and business logic, other concerns will be implemented separately (by different developers).
- Spring implementation of AOP is based on open source AOP project "Aspect/J".

AOP Terminologies

- Aspect:
  - Additional functionality (cross-cutting concern) other than business logic.
  - In spring, implemented as a separate class annotated with @Aspect.
- Advice:
  - The operation to be performed to handle certain cross-cutting concern
  - In spring, it is implemented as methods in aspect class.
  - There are four types of advices given by different annotations (on advice methods):
    - @Before - pre-processing logic
    - @After - post-processing logic
    - @Around - pre-processing and post-processing logic
    - @AfterThrowing - post-processing in case business logic method throws exception
- Target:
  - The object on which business logic method is called.
- Proxy:
  - This object is internally created by the spring container to call the advices and the actual business logic.
- JoinPoint:
  - Represents the point in the application, where advice is to be applied.
  - In spring it is an "JoinPoint" object which contains information of the business logic on which advice is applied. It also contains information about the object on which business logic method is called.
- Pointcut:
  - combination of one or more join points where advice is to be applied
  - In spring PointCut is implemented as an empty method with @PointCut annotation
- Advisor:
  - Group of advice and pointcut into a single unit called as "Advisor".
  - This object is internally passed to proxy factory to create the proxy object.

Spring AOP Example

- step 1. Implement an interface IAccount and spring bean class Account.

```
//IAccount.java
public interface IAccount {
	public void deposit(double amount);
	public void withdraw(double amount);
}
//Account.java
public class Account {
	private int id;
	private double balance;
	// constructors
	// getter/setters
	// toString()
	public void deposit(double amount) {
		this.balance += amount;
	}
	public void withdraw(double amount) {
		this.balance -= amount;
	}
}
```

- step 2. Implement cross-cutting concern in separate aspect class.

```
// AccountAspects.java
```

```
@Component
@Aspect
public class AccountAspects {
	@Before("execution (* pkg.Account.set*(..))")
	public void logBefore(JoinPoint pt) {
		System.out.println("BEFORE : " + pt.getSignature());
	}
	@After("execution (* pkg.Account.get*(..))")
	public void logBefore(JoinPoint pt) {
		System.out.println("AFTER : " + pt.getSignature());
	}
	@Around("execution (* pkg.Account.withdraw(..))")
	public Object monitorTransaction(ProceedingJoinPoint pt) throws Throwable {
		IAccount acc = (IAccount)pt.getTarget();
		System.out.println("Balance Before : " + acc.getBalance());
		Object res = pt.proceed();
		System.out.println("Balance After : " + acc.getBalance());
		return res;
	}
}
```

- In order to activate aspect classes XML configuration file should contain.

```
<aop:aspectj-autoproxy/>
```

- Instead of XML configuration file, Java configuration class can be annotated with @EnableAspectJAutoProxy.

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {
	// ...
}
```

- When methods are called on Account object, corresponding advice methods are executed from AccountAspects class.

Spring Hibernate Integration

- Spring simplifies Hibernate ORM programming by abstracting boilerplate code into pre-defined beans and annotations.
- Spring helps creating SessionFactory, transaction management and simplifying configuartion of Hibernate.

Spring Hibernate Integration steps

- step 1. In pom.xml add dependencies for hibernate-core, mysql-connector, spring-context and spring-orm.
- step 2. In spring bean configuartion file (or Java config class) create dataSource, sessionFactory and transactionManager beans. Also set transactionManager as default transaction manager.

```
<context:component-scan base-package="com.sunbeaminfo.sh"/>

<bean id="mysqlDataSource"
	class="org.springframework.jdbc.datasource.DriverManagerDataSource">
	<property name="driverClassName"
		value="com.mysql.cj.jdbc.Driver" />
	<property name="url"
		value="jdbc:mysql://localhost:3306/test?useSSL=false" />
```

```
		<property name="username" value="nilesh" />
		<property name="password" value="****" />
	</bean>

	<bean id="mysqlSessionFactory"
		class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
		<property name="dataSource" ref="mysqlDataSource" />
		<property name="hibernateProperties">
			<props>
				<prop
key="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</prop>
				<prop key="hibernate.show_sql">true</prop>
			</props>
		</property>
		<property name="packagesToScan" value="com.sunbeaminfo.sh.entities"/>
	</bean>

	<bean id="mysqlTransactionManager"
class="org.springframework.orm.hibernate5.HibernateTransactionManager">
		<property name="sessionFactory" ref="mysqlSessionFactory"/>
	</bean>

	<tx:annotation-driven transaction-manager="mysqlTransactionManager"/>
```

- step 3. Implement Hibernate entity classes and setup relations among them (as required).
- step 4. Implement repository interface and classes.

```
@Repository
public class CustomerRepositoryImpl implements CustomerRepository {
	@Autowired // injects "mysqlSessionFactory" declared in config file
	private SessionFactory sessionFactory;
	public Customer getCustomer(int id) {
		Session session = sessionFactory.getCurrentSession();
		return session.get(Customer.class, id);
	}
	public void addCustomer(Customer c) {
		session.persist(c);
	}
	// ...
}
```

- step 5. Implement service interface and classes. Also setup transaction context.

```
@Service
public class CustomerServiceImpl implements CustomerService {
	@Autowired
	private CustomerRepository customerDao;
	@Transactional
	public Customer getCustomer(int id) {
		return customerDao.getCustomer(id);
	}
	@Transactional
	public void addCustomer(Customer c) {
		customerDao.addCustomer(c);
	}
```

```
        // ...
    }
```

- step 6. Get service bean object and invoke methods of service class.

@Transactional annotation

- @Transactional annotation can be applied to method or class level. If applied at class level, it is applicable to all methods in the class.
- Usually @Transactional is added into service class. Typically service class deals with one or more DAO classes. Using @Transactional for service class, ensure that all DAO methods are accessed in same transaction.
- @Transactional annotation have two important properties.
  - propogation: Represents with nested transaction propgation.
  - isolation: Represents with transaction isolation levels.

Transactional propogation values

- MANDATORY: Support a current transaction, throw an exception if none exists.
- NESTED: Execute within a nested transaction if a current transaction exists, behave like REQUIRED otherwise.
- NEVER: Execute non-transactionally, throw an exception if a transaction exists.
- NOT_SUPPORTED: Execute non-transactionally, suspend the current transaction if one exists.
- REQUIRED: Support a current transaction, create a new one if none exists.
- REQUIRES_NEW: Create a new transaction, and suspend the current transaction if one exists.
- SUPPORTS: Support a current transaction, execute non-transactionally if none exists.

Transactional isolation

- The typical issues while concurrent transactions are dirty reads, non-repeatable reads and phantom reads.
- Possible @Transactional isolation values are as follows.
  - DEFAULT: Use the default isolation level of the underlying datastore.
  - READ_COMMITTED: A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
  - READ_UNCOMMITTED: A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur.
  - REPEATABLE_READ: A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
  - SERIALIZABLE: A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented.

Spring Data Templates

- To reduce boilerplate code and simplify operations, spring provide some template classes.
- There are several template classes available for different tasks e.g. JdbcTemplate, HibernateTemplate, JpaTemplate, JmcTemplate, RestTemplate, MongoTemplate, etc.

HibernateTemplate

- HibernateTemplate is wrapper/template for spring-hibernate integration. It also automate transaction management.
- However HibernateTemplate is not recommended in Spring 3.x and onwards, because it tightly couples hibernate code with spring framework.

- Note that steps for using HibernateTemplate are similar to that of Spring Hibernate integration.
- step 1. In pom.xml add dependencies (as in spring-hibernate integration).
- step 2. In spring bean configuartion file (or Java config class) create dataSource and sessionFactory beans (as in spring-hibernate integration). Also create hibernateTemplate bean.

```
<bean id="mysqlHibernateTemplate"
class="org.springframework.orm.hibernate5.HibernateTemplate">
	<property name="sessionFactory" ref="mysqlSessionFactory"/>
</bean>
```

- step 3. Implement Hibernate entity classes and setup relations among them (as required).
- step 4. Implement repository interface and classes.

```
@Repository
public class CustomerRepositoryImpl implements CustomerRepository {
	@Autowired // injects "mysqlHibernateTemplate" declared in config file
	private HibernateTemplate hibernateTemplate;
	public Customer getCustomer(int id) {
		Session session = sessionFactory.getCurrentSession();
		return hibernateTemplate.get(Customer.class, id);
	}
	public void addCustomer(Customer c) {
		hibernateTemplate.persist(c);
	}
	// ...
}
```

- step 5. Implement service interface and classes.

```
@Service
public class CustomerServiceImpl implements CustomerService {
	@Autowired
	private CustomerRepository customerDao;

	public Customer getCustomer(int id) {
		return customerDao.getCustomer(id);
	}

	public void addCustomer(Customer c) {
		customerDao.addCustomer(c);
	}
	// ...
}
```

- step 6. Get service bean object and invoke methods of service class.

JdbcTemplate

- JdbcTemplate is wrapper/template for spring-jdbc integration.
- It simplifies JDBC programming by removing boilerplate code like create connection, create PreparedStatement and execute them. Also it avoid repetitive mapping of database row with Java object and vice-versa.
- JdbcTemplate approach is sometimes preferred as it is lightweight and doesn't have overheads of ORM.

- step 1. In pom.xml add dependencies for mysql-connector, spring-context and spring-jdbc.
- step 2. In spring bean configuartion file (or Java config class) create dataSource, jdbcTemplate beans. Also set transactionManager as default transaction manager.

```
<context:component-scan base-package="com.sunbeaminfo.sh"/>

<bean id="mysqlDataSource"
	class="org.springframework.jdbc.datasource.DriverManagerDataSource">
	<property name="driverClassName"
		value="com.mysql.cj.jdbc.Driver" />
	<property name="url"
		value="jdbc:mysql://localhost:3306/test?useSSL=false" />
	<property name="username" value="nilesh" />
	<property name="password" value="****" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
	<property name="dataSource" ref="mysqlDataSource"/>
</bean>
```

- step 3. Implement repository interface and class. Implement RowMapper class (can be implemented as nested class).

```
@Repository
public class BookDaoJdbcImpl implements BookDao {

	public static class BookRowMapper implements RowMapper<Book> {
		@Override
		public Book mapRow(ResultSet rs, int rowNum) throws SQLException {
			int id = rs.getInt("ID");
			String name = rs.getString("NAME");
			String author = rs.getString("AUTHOR");
			String subject = rs.getString("SUBJECT");
			double price = rs.getDouble("PRICE");
			return new Book(id, name, author, subject, price);
		}
	}

	@Autowired
	private JdbcTemplate jdbcTemplate;

	@Override
	public void addBook(Book b) {
		String sql = "INSERT INTO BOOKS(ID,NAME,AUTHOR,SUBJECT,PRICE)
VALUES(?,?,?,?,?)";
		Object params[] = new Object[] {
			b.getBookId(),
			b.getName(),
			b.getAuthor(),
			b.getSubject(),
			b.getPrice()
		};
		jdbcTemplate.update(sql, params);
	}
```

```
	@Override
	public Book getBook(int id) {
		String sql = "SELECT ID, NAME, AUTHOR, SUBJECT, PRICE FROM BOOKS WHERE
ID=?";
		Object params[] = new Object[] {
			id
		};
		Book b = jdbcTemplate.queryForObject(sql, params, new BookRowMapper());
		return b;
	}

	@Override
	public List<Book> getAllBooks() {
		String sql = "SELECT ID, NAME, AUTHOR, SUBJECT, PRICE FROM BOOKS";
		Object params[] = new Object[] {
		};
		List<Book> list = jdbcTemplate.query(sql, params, new BookRowMapper());
		return list;
	}
 }
```

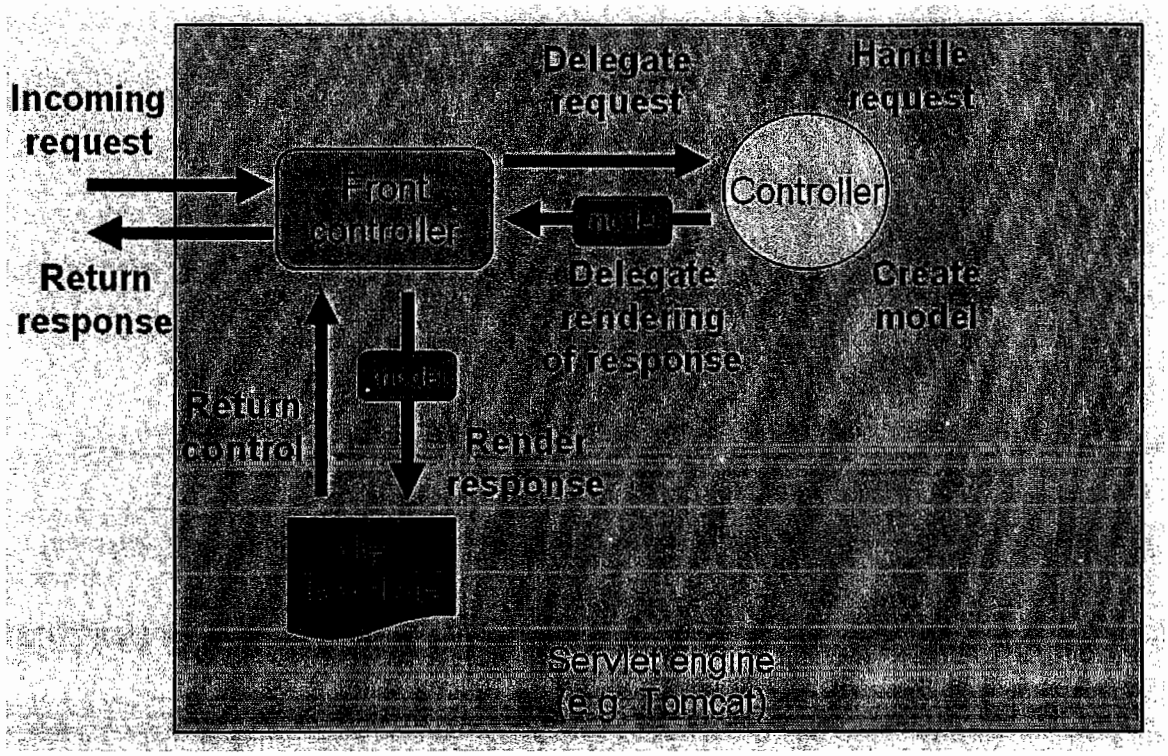
- step 4. Implement service layer class (optional for simple code), which simply wraps repository layer.
- step 5. Get service bean object and invoke methods of service class.

Spring Web MVC framework

- MVC framework is provided under Spring for developing web applications.

Understanding Spring MVC

- MVC frameworks has three components M-Model, V-View and C-Controller.



Controller

- Spring MVC framework provides a main controller through which all requests are routed. It is called as "Front Controller" and is implemented in a class "DispatcherServlet".
- Front Controller is responsible for finding appropriate request handler method when request arrives from client. Also it finds appropriate view page when user defined controller returns (with the help of viewResolver).
- In spring developer can implement his own controller classes, which typically invokes business logic classes (@Service) and then navigate to the views based on business logic result.
- Methods in these controller are called as "request handler methods". They are annotated with @RequestMapping(). Usually there is single method in whole application with particular @RequestMapping("/...") annotation.
- Request handler method signature is very flexible. It can take various arguments as well as various return types (refer spring framework manual - topic: request handler method).

View

- Contains presentation logic and typically implemented as JSP pages. (Also new view engines like Thymeleaf are supported, but beyond scope of this book).
- These pages can contain JSTL and third party tags as well as spring tags.
- Example form tag as follows:
- <sf:form action="something" modelAttribute="command">
- ...
  </sf:form>
- In spring form tag, there is modelAttribute which refer to a request attribute. It is a model/POJO object and data from that model object will be shown in controls within form tag.

Model

- Contains data to be carried between view and controller.
- They are simply POJO objects.
- From view data can be collected in POJO object by simply making it as argument of request handler method. In request handler method, these objects can be annotated as @ModelAttribute.

WebApplicationContext

- Bean creation and dependency injection is responsibility of application context and it represents spring container.
- In web mvc application, this job is done by "WebApplicationContext".
- These application contexts can be created & maintained hierarchically.
- WebApplicationContext configuration is written in a XML file.
- When application is deployed, ContextLoaderListener is executed, which creates root WebApplicationContext and tie it with ServletContext of the application. This is main application context in web application and is configured through XML file whose path is given in context parameter - contextConfigLocation.
- After this DispatcherServlet (front controller) is loaded (because it is declared as "load-on-startup" servlet).
- This also creates another WebApplicationContext as child of main WebApplicationContext. It is responsible for basic MVC functionality.
- Default name for its configuration file is "servletname-servlet.xml". However, different name can be also be provided using servlet init parameter.
- For simple web application this WebApplicationContext is sufficient. However for the advanced features like security root WebApplicationContext (created by ContextLoaderListener) is required.

Spring Web MVC Localization

- Spring beans supports Localization feature i.e. web page can be shown in multiple languages based on browser locale.
- The locale is represented as four letter string including country and language. For example, en-US, en-GB, fr-FR, hi-IN, etc.
- For this we need to store text for different locales in different properties files. Note that Unicode strings should be converted ASCII format (using JDK native2ascii tool).
- Steps for implementing Localization in spring is as follows.
- step 1. Write all necessary strings in required locale files. Note that file name ends with locale name.

```
# WEB-INF/mesg_en.properties
app.title = Nilesh Ghule
WEB-INF/mesg_hi.properties
app.title = \u0928\u093f\u0932\u0947\u0936 \u0918\u0941\u0932\u0947
```

step 2. In controller class, "locale" can be accessed by request handler method argument of type java.util.Locale (if required).

```
java.util.Locale.
@RequestMapping(value = "/abc")
public String method(Locale locale, Model model) {
      System.out.println("Locale : " + locale);
      // ...
}
```

- step 3. In JSP pages, use tag "message" to display text from properties file.

```
<spring:message code="app.title"/>
```

- step 4. Configure resourceBundle bean in bean configuartion file. In order to keep same locale throughout the web-site use localeResolver bean. Optionally to change the current locale based on request parameter value, add LocaleChangeInterceptor.

```
 <bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
      <property name="basename" value="WEB-INF/mesg" />
      <property name="defaultEncoding" value="UTF-8" />
 </bean>

 <bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
      <property name="defaultLocale" value="en" />
      <property name="cookieName" value="myAppLocaleCookie"/>
      <property name="cookieMaxAge" value="3600"/>
 </bean>

 <mvc:interceptors>
      <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
            <property name="paramName" value="locale" />
      </bean>
 </mvc:interceptors>
```

- In above configuartion
  - messageSource bean is configured to enable i18n for our application.
    - "basename" property is used to provide the location of resource bundles. "classpath:messages" means that resource bundles are located in the classpath and follows name pattern as messages_{locale}.properties.
    - "defaultEncoding" property is used to define the encoding used for the messages.
  - "localeResolver" bean is used to keep same locale throughout the web-site (for that user).
    - The CookieLocaleResolver bean is used to set a cookie in the client response, so that further requests can easily recognize the user locale.
    - If application maintains user sessions, then SessionLocaleResolver bean can also be used as localeResolver.
    - If we don't register any "localeResolver", AcceptHeaderLocaleResolver will be used by default, which resolves user locale by the "accept-language" header in the client HTTP request.
  - To change locale programmatically (based on user request), interceptor bean is used.
    - The LocaleChangeInterceptor interceptor bean is configured to intercept the user request and identify the user locale. The parameter name is configurable.
    - In above configuration request parameter name for locale as "locale".

Spring MVC Validation

- It is recommended to implement client as well as server side validation for user form data.
- Client side validation is done using HTML5 controls and using Java script libraries.
- The steps for server side validation is as follows.
- step 1. Use validator annotations on Model/POJO class fields.
  - Hibernate validator annotations: @NotEmpty, @Length, @Range, @Email, @DateTimeFormat, @URL
  - Javax Annotations: @NotBlank, @Size, @Min, @Max, @Pattern

- step 2. The validation messages can be provided into annotation itself or into messages.properties file (resource bundle) as follows.
  - ConstraintName.ModelAttrName.fieldName = validation message.
  - e.g. NotEmpty.user.username = username cannot be blank.
- step 3. If resource bundle file is used, then resourceBundle bean should be declared into spring bean configuration file.

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
	<property name="basename" value="WEB-INF/mesg" />
	<property name="defaultEncoding" value="UTF-8" />
</bean>
```

- step 4. Implement corresponding request handler method.
  - Anotate model object as @ModelAttribute as well as @Valid. @Valid will validate object fields as per validator annotations.
  - The "BindingResult" argument must be passed immediately after model object. It collects the result of validation. If error found, redirect to input page or any page on which error to be displayed.

```
@RequestMapping("/register")
public String register(Model model, @Valid @ModelAttribute("user") Login login,
BindingResult res) {
	if(res.hasErrors()) {
		model.addAttribute("status", "Form submission has errors.");
		return "register";
	}
	// process model object data
	model.addAttribute("status", "Form submission is successful.");
	return "register";
}
```

- step 5. In view.jsp page display error using <sf:error path="fieldName" cssClass="error"/>. Optionally all error messages can be displayed using <sf:errors path="*" cssClass="error"/>.

Spring Custom Validations

- Other than built-in annotations, custom validation logic can be implemented with following steps.
- step 1. Implement a bean class (@Component) inherited from "org.springframework.validation.Validator".
  - Implement supports() to check if Model class is supported by this validator.
  - Implement validate() to validate object and add errors into Errors object, if any. The value can be rejected using helpers like ValidationUtils.rejectIfEmpty() or err.rejectValue("fieldName", "MsgBundle_KeyName", "Default message").
- Autowire validator into Controller class.
- Attach validator for Controller class in @InitBinder method.

```
@Autowired
private CustomerValidator custValidator;

@InitBinder
public void initValidator(WebDataBinder binder) {
	binder.addValidators(custValidator);
}
```

- Use @Valid for model object and handle error messages (as discussed earlier).

Spring REST services

- Traditionally web services are implemented as SOAP (Simple Object Access Protocol) web services. However due to client side proxies, XML parsing and other stuff, they were not suitable for small devices (like mobiles).
- The REST service is modern way of implementing web services.

REST Protocol

- REST stands for Representational State Transfer.
- Object state representation can be transferred between server and client in XML or JSON format. REST web service can consume and produce XML as well as JSON format.
- It can be simply accessed using URL. Unlike SOAP services it doesn't need proxy objects.
- These services can be easily used with Mobile, Java Script or Desktop clients irrespective of language of implementation.
- Also they are easy to test using browsers or applications like POSTMAN.
- Most commonly REST services work on top of HTTP protocol and use HTTP GET, POST, PUT and DELETE methods.
    - GET: To fecth one or more objects from server.
        - http://localhost:8080/appln/books (Fetch all books)
        - http://localhost:8080/appln/books/22 (Fetch book with id 22)
    - DELETE: To delete object/row from server.
        - http://localhost:8080/appln/books/22 (Delete book with id 22)
    - POST: To insert new object on server side.
        - http://localhost:8080/appln/books (Add book whose data is present in request body in XML/JSON format.)
    - PUT: To update object on server side.
        - http://localhost:8080/appln/books/22 (Update book with id 22. The modified data is present in request body in XML/JSON format.

REST services using Spring

- Spring have built-in support for REST services using some additional jars for XML and/or JSON parsing.
- Steps for the Spring REST application are as follows.
- step 1. Add dependencies in pom.xml of jackson-databind and jackson-dataformat-xml along with other spring-mvc dependencies.
- step 2. Implement request handler methods.
    - Write request handler method with proper mapping & HTTP request method.
    - If object is argument for request handler method, it should be annotated with @RequestBody.
    - The return value of request handler method, should be @ResponseBody. Alternatively the controller clas can be annotated as @RestController (instead of @Controller).
    - Also @RequestMapping(method=RequestMethod.GET, ...) can be replaced by @GetMapping. Similarly @PostMapping, @PutMapping and @DeleteMapping is also available.

```
@Controller
public class BookRestController {
	@Autowired
	private BookService service;
	@RequestMapping(value="/books/{bkid}", method=RequestMethod.GET)
	public @ResponseBody Book findBook(@PathVariable("bkid") int id) {
		return service.getBook(id);
	}
```

```
	@RequestMapping(value="/books/{bkid}", method=RequestMethod.DELETE)
	public @ResponseBody String delBook(@PathVariable("bkid") int id) {
		try {
			service.deleteBook(id);
			return "success";
		} catch(Exception e) {
			e.printStackTrace();
			return "failed";
		}
	}
	@RequestMapping(value="/books", method=RequestMethod.POST)
	public @ResponseBody String addBook(@RequestBody Book b) {
		try {
			service.addBook(b);
			return "success";
		} catch(Exception e) {
			e.printStackTrace();
			return "failed";
		}
	}
}
```

- step 3. Configure converter beans.
  - By default returned object is converted into JSON format (if corresponding dependencies are added in the project).
  - However by adding xmlConverter and jsonConverter beans, the output can be generated in desired format.
  - One of these converters is used while handling request (by RequestMappingHandlerAdapter).
  - If both converters are configured, the output format is driven by request headers (Accept or Content-Type) or request handler annotation properties (produces or consumes).

```
<bean id="jsonConverter"
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
<bean id="xmlConverter"
class="org.springframework.http.converter.xml.MappingJackson2XmlHttpMessageConverter"/>
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapte
r">
	<property name="messageConverters">
		<list>
			<ref bean="jsonConverter"/>
				<ref bean="xmlConverter"/>
		</list>
	</property>
</bean>
```

- step 4: Testing request handler methods.
  - GET methods can also be tested using simple browser.
    - Visit link http://localhost:8080/appln/books/22.
  - Some tools like Postman are used to test services with all possible methods. This tool can also generate client code in many programming/scripting languages.
- step 5: Implement Spring Rest client (optional).

- Rest services can be consumed by JS libraries in web pages. They can also be invoked from mobile applications.
- Spring application can also consume Rest services using RestTemplate class.
- Note that client project should have dependencies like spring-web and jackson-databind (json libraries).

```
RestTemplate template = new RestTemplate();

String url1 = "http://localhost:8080/appln/books/11";
Book b1 = template.getForObject(url1, Book.class);
System.out.println(b1);

String url2 = "http://localhost:8080/appln/books";
Book b2 = new Book(101, "Atlas Shrugged", "Any Rand", "Novell", 523.34);
String status = template.postForObject(url2, b2, String.class);
System.out.println("Book Added: " + status);
```

Spring Security

- Authentication & Authorization
  - Authentication is processing credentials of the user and validating it. It provides identity of the user.
  - Authorization checks if authenticated user has permissions to access intended resource. It deals with "Role Management" in the web application.
- step 1. Database Design
  - Table1: USERS: USERNAME, PASSWORD, ENABLED.
  - Table2: USER_ROLES: ROLEID, USERNAME, ROLE.
- step 2. web.xml Configuration

```
<filter>
	<filter-name>springSecurityFilterChain</filter name>
	<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
	<filter-name>springSecurityFilterChain</filter-name>
	<url-pattern>/*</url-pattern>
</filter-mapping>
```

- step 3. Spring (4.3) XML based configuartion

```
<security:authentication-manager>
	<security:authentication-provider>
		<security:jdbc-user-service data-source-ref="dataSrc" users-by-username-
query= "select username,password, enabled from users where username=?" authorities-by-
username-query="select username, role from user_roles where username=?"/>
	</security:authentication-provider>
</security:authentication-manager>

<security:http auto-config="true" use-expressions="true">
	<security:intercept-url pattern="/admin/**" access="hasAnyRole('ROLE_ADMIN')" />
	<security:intercept-url pattern="/book/**" access="hasAnyRole('ROLE_USER')" />
	<security:form-login login-page="/login" default-target-url="/auth"
	authentication-failure-url="/login" />
	<security:logout logout-success-url="/logout" />
</security:http>
```

- step 4. Create login JSP page.

```
<form ... action="j_spring_security_check">
      UserName: <input type="text" name="j_username"/> <br/>
      Password: <input type="text" name="j_password"/> <br/>
      <input type="submit" value="Sign In"/>
</form>
```

- step 5. Implement Controller classes with appropriate RequestMapping url.
    - Note that these URL patterns will be basis for authorization as configured in <security:intercept-url>.

```
@Controller
@RequestMapping("/book")
public class BookShopController {
      // ...
}

@Controller
@RequestMapping("/admin")
public class AdminController {
      // ...
}
```

Sonali Sagar.

DBT

+ File System:
- File: Collection of data or information on a storage device.
- File = Contents (Data) + Information (Metadata).
- Data is stored in one/more data blocks on FS.
- Metadata is stored in File Control Block (FCB). Also known as i-node.
- File type [- d l p s c b]
- Size, Number of blocks
- User & Group info
- File permissions: rwx rwx rwx
- Link count
- Time stamps: Creation, Modified, Accessed.
- Information about data blocks
- File System is way of organizing data on the disk e.g. FAT, NTFS, EXT4, XFS, HFS, ...
- File System : [Boot block | Super block | iNode list | Data blocks]
- Boot block : booting related programs: bootloader
- Super block : information about partition - part size, block size, label, total number of data blocks, number of free data blocks, which data blocks are free.
- inode list : The FCB/Inode for each file.
- Data blocks : One/more data blocks for each file.

+ DBMS:
- It is collection of programs that allows inserting, updating, deleting and processing the records.
- e.g. Excel, DBase, Tally, Foxpro, ...
- These are readymade software which allows you to manage the data.
- Few softwares allows programming in it e.g. Excel macros, ...

+ RDBMS:
- Advanced DBMS that allow storing/processing data in tabular format, where relationship is maintained across the processes. RDBMS also support programming language support -- SQL & PL-SQL and designed to access securely in client-server environment using multiple different clients.
- e.g. Informix, Oracle, Sybase, MySQL server, Ingres, PostgresSQL, Unify, DB2, CICS, TELON, MS Access, Paradox, Vatcom SQL, MSSQL server, MySQL (MariaDB), ...

+ DBMS vs RDBMS:

	DBMS	RDBMS
1.	Fields	Columns/Attributes
2.	Record	Row/Entity
3.	File	Table, Relation
4.	Relation between two files is done programmatically.	Relation between two tables is given while table creation -- primary key/foreign key.
5.	Lot of programming needed.	Database engine does lot of processing. Less programming needed.
6.	High network traffic -- whole DB file is transferred over network.	Low network traffic -- only fetched records will be transferred.
7.	Processing is on client machine.	Processing is on server machine.

8.	Slower -- Do not support huge data.	Faster -- Support huge data.
9.	Client-Server architecture is not supported.	Most of RDBMS support Client-Server architecture.
10.	Not suitable for multi-user.	Suitable for multi-user.
11.	File level locking.	Row(record) level locking. Few RDBMS treat each row as file.
12.	Distributed database is not supported.	Most of the RDBMS support distributed database.
13.	DBMS does not have any security features. Depends on OS features. Data can be accessed using that DBMS only.	Security is built-in feature of RDBMS. RDBMS support multiple security levels. OS security, RDBMS login security, Command level security, Object level security.

+ Important RDBMS:

1. Informix: Fastest RDBMS. But program should be in assembly.
2. Oracle: Most popular RDBMS. Supported on multiple platforms (113).
3. Sybase: Decreasing usage. Taken over by SAP.
4. MS SQL Server: Very good RDBMS. Only for windows.
5. Ingres, PostgresSQL & Unify: Open source RDBMS. Mostly on UNIX/Linux.
6. DB2 (IBM), CICS, TELON, IDMS: Only on mainframe computers.
7. MSAccess, Paradox, Vatcom SQL: Single user database (not client-server).
8. MySQL, MariaDB : Opensource RDBMS.

+ MySQL:

- Was launched by a Swedish company in 1995. The whole source code implemented in C/C++.
- The name MySQL is in name of daughter founder member Michael Widenius i.e. Myia.
- MySQL is open source RDBMS.
- It is most widely used open-source client server model RDBMS.
- MySQL occupies 42% of world open-source database market.
- MySQL is part of widely used LAMP open source web appln software stack.

Stack: OS/Platform + Web Server + Database + Server-side Programming
LAMP: Linux + Apache + MySQL + PHP
WAMP: Windows + Apache + MySQL + PHP
MAMP: Mac OS X + Apache + MySQL + PHP

- Most prominent users: wordpress, google (not for search engine), facebook, twitter, youtube, joomla, drupal, ...
- Oracle/MySQL have best tools for software development. It makes programming & software development easier.
- Sun Microsystems acquired MySQL in 2008. Oracle takes over Sun Microsystems in 2010.
- MySQL Enterprise Edition: Additional modules for commercial use and closed-source.

- Maria DB: Fully open-source database started as a branch of MySQL.
- No close source modules like in MySQL Enterprise Edition -- all components are under GPL.
- Designed to be drop-in replacement for MySQL.
- Nowadays, xAMP stacks comes in with MariaDB. Also with systems like CentOS.
- This is not exactly identical to MySQL. MySQL support additional database engines.

+ SQL: Structured Query Language:
- Commonly pronounced as "Sequel".
- Originally started by IBM (1975-77) known as RQBE (Relational Query By Example).
- Now it conforms to ANSI standard -- 1 character = 1 byte.
- It also conforms to ISO standard -- Quality assurance.
- It is common for all RDBMS. However each RDBMS have some extended SQL features, which are database specific. PL (programming) syntax will change considerably from RDBMS to RDBMS.
- SQL is case insensitive language.
- SQL statements are referred as SQL queries.
- Mainly 5 types of queries in SQL:

A. DDL - Data Definition Language
- CREATE ...
- ALTER ...
- DROP ...
- RENAME ... *
- DESCRIBE ... *

B. DML - Data Manipulation Language
- INSERT ...
- UPDATE ...
- DELETE ...

C. DQL - Data Query Language
- SELECT ...

D. DCL - Data Control Language
- GRANT ...
- REVOKE ...

E. DTL / TCL - Data Transaction Language / Transaction Control Language
- Transaction is set of queries executed as a single unit. If any of the queries from that unit fails, remaining queries effect will be reversed.
- SAVEPOINT ...
- COMMIT ...
- ROLLBACK ...

+ MySQL/MariaDB:
- mysql -V

MariaDB 5.5.52 <-- CentOS
mysql 5.7.19 <-- Ubuntu
- To start mysql server (from server terminal):
- sudo systemctl start mariadb
- sudo systemctl start mysql
- To stop mysql server (from server terminal):
- sudo systemctl stop mariadb
- sudo systemctl start mysql
- To start mysql server during booting:
- sudo systemctl enable mariadb
- sudo systemctl start mysql
- To stop starting mysql server during booting:
- sudo systemctl disable mariadb
- sudo systemctl start mysql
- To check status of mysql server:

- sudo systemctl status mariadb
- sudo systemctl start mysql
- The default port for mysql db is 3306.
    - netstat -t -l -n
- To check if mysql server is running:
    - ps -e
    --> look for process "mysqld"

- To connect mysql database (from client terminal):
mysql -h <hostname> -u <username> -p
e.g.
A. mysql -h localhost -u root -p --> -p is prompt for password
B. mysql -u root -p --> by default connect to mysql server on localhost
C. mysql -u root -proot --> password is "root"; not prompted.
D. mysql -u root -p db_name --> login and use given db schema as default.
E. mysql -u sunbeam -p dbname

- To see list of databases/schemas:
    - SHOW DATABASES;
- To create a database:
    - CREATE DATABASE DB_NAME;
- To create user:
    - CREATE USER 'user'@'hostname' IDENTIFIED BY 'password';
- To give permissions to user on a database:
    - GRANT ALL PRIVILEGES ON DB_NAME.* TO 'user'@'hostname';
- To use db:
    - USE DB_NAME;
- To see tables in current database;
    - SHOW TABLES;

+ DCL commands:
- Gives full permissions on database.
    - GRANT ALL PRIVILEGES ON DB_NAME.* TO 'user'@'hostname';
- Give permissions on table:
- GRANT SELECT ON DBDA_DB.BOOKS TO 'user'@'hostname';
- GRANT DELETE ON DBDA_DB.BOOKS TO 'user'@'hostname';
- GRANT INSERT ON DBDA_DB.BOOKS TO 'user'@'hostname';
- GRANT UPDATE ON DBDA_DB.BOOKS TO 'user'@'hostname';
- Remove permissions on table:
- REVOKE DELETE ON DBDA_DB.BOOKS TO 'user'@'hostname';
- REVOKE UPDATE ON DBDA_DB.BOOKS TO 'user'@'hostname';

+ DDL commands:
- CREATE TABLE TABLE_NAME (COLNAME1 DATATYPE, COLNAME2 DATATYPE, ...);
e.g.
CREATE TABLE BOOKS (ID INT(4), NAME VARCHAR(50), AUTHOR VARCHAR(50), SUBJECT VARCHAR(50), PRICE DOUBLE(7,2));

- Naming Conventions:

- Maximum name length should be 30 chars long.
- Can contain alphabets, digits and special symbols : _, $ & #.
- The name must start with alphabet.

mysql2007 -- valid name
2007mysql -- invalid name
- While using # special character name should be enclosed in `backquotes` e.g. `sunbeam#`
- As per standard ANSI names are case insensitive. However it may change for few DB & OS. e.g. In Linux the TABLE name is case-sensitive.

+ Data Types in MySQL:
+ Numeric: Length specifies number of chars/digits to be displayed.
Integer values:

TINYINT, BOOL/BOOLEAN	1 byte	-128 to 127
SMALLINT	2 bytes	-32768 to 32767
MEDIUMINT	3 bytes	-8388608 to 8388607
INT, INTEGER	4 bytes	-2147483648 to 2147483647
BIGINT	8 bytes	-9223372036854775808 to 9223372036854775807
BIT(M) – bitfields	M/8 bytes	1 to 64 bits

* Except BIT all data types support ZEROFILL & UNSIGNED attribute.
- ZEROFILL store preceding zeros for a numbers.
INT(5) -- 32 => 32
INT(5) ZEROFILL -- 32 => 00032
- UNSIGNED doesn't allow -ve values. So one bit more available for data storage, hence positive range increases.
* TRUE is synonym for 1 and FALSE is synonym for 0.
* For BOOL values, 0 is false condition while non-zero is true condition.
Approximate precision:

FLOAT	4 bytes	-3.402823466E+38 to -1.175494351E-38, 0, 1.175494351E-38 to 3.402823466E+38
DOUBLE,REAL	8 bytes	-1.7976931348623157E+308 to -2.2250738585072014E-308, 0, 2.2250738585072014E-308 to 1.7976931348623157E+308

Fixed precision:

DECIMAL,NUMERIC	as per precision	Store number as string -- for accurate precision

+ Date & Time:

DATE	3 bytes	'1000-01-01' to '9999-12-31'

		Stored as number of days from min range.
TIME	3 bytes	'-838:59:59' to '838:59:59' Stored as number.
DATETIME	8 bytes	'1000-01-01 00:00:00' to '9999-12-31 23:59:59' Combines date and time.
TIMESTAMP	4 bytes	'1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' Stored as number seconds from epoch time.
YEAR(4)	1 byte	1901 to 2155

* In Oracle Db: Dates missing from 4-OCT-1582 to 15-OCT-1582 (Calender change).
* TIMESTAMP used to keep track of insertion/updation time of record in database.

+ String:

CHAR(M)	M * w bytes (w depends on char-set)	Max 255 chars. Fixed storage.
BINARY(M)	M bytes	(ASCII chars only) Max 255 bytes. Fixed storage.
VARCHAR(M) VARBINARY(M)	L+1 or L+2 bytes (Stores length)	Max len 65535. Efficient storage.
TINYBLOB TINYTEXT	L + 1 bytes	Max len 255
BLOB TEXT	L + 2 bytes	Max len 65535
MEDIUMBLOB MEDIUMTEXT	L + 3 bytes	Max len 16777215
LONGBLOB LONGTEXT	L + 4 bytes	Max len 4294967295
ENUM('v1','v2')	1 or 2 bytes	Max values 65,535
SET(...)	1, 2, 3, 4, or 8 bytes	Max members 64

* CHAR(M) & BINARY(M) occupies fixed size on disk. If CHAR(100) columns store value of 10 chars only, still it occupies 100 locations on disk.
* Can specify character set for char values.

```
CREATE TABLE temp (c1 VARCHAR(20) CHARACTER SET utf8, c2 TEXT ASCII);
- ASCII is shorthand for CHARACTER SET latin1
- UNICODE is shorthand for CHARACTER SET ucs2
CREATE TABLE test1 (c1 CHAR(6), c2 VARCHAR(6));
- c1 -- 'AB' =>'AB....'
- c2 -- 'AB' => 2 'AB'
```

CREATE TABLE test (c1 CHAR(6) CHARACTER SET BINARY, c2 BINARY(6));
- c1 & c2 both are same in data type.

	VARCHAR	TEXT
1.	Storage is inline in the record/row.	Storage is outside the record and record contains pointer to text data.
2.	Can be used for indexing (faster search).	Cannot be used for indexing.
3.	Faster access.	Slower access.

+ DML - INSERT query:
- INSERT INTO BOOKS VALUES(1001, 'Let Us C', 'Kanetkar', 'C', 234.56);
- The values must be in same sequence in which it is declared in table.
- INSERT INTO BOOKS (ID,NAME,SUBJECT,AUTHOR,PRICE)
VALUES('1002', 'Pointer In C', 'C', 'Kanetkar', 342.20);
- More readable way to insert query.
- INSERT INTO BOOKS (ID,NAME,PRICE) VALUES('1002', 'Sherlock', 342.20);
- The remaining columns get NULL values.
- INSERT INTO NEW_BOOKS (ID, NAME, AUTHOR, SUBJECT, PRICE) SELECT ID, NAME, AUTHOR, SUBJECT, PRICE FROM BOOKS WHERE PRICE > 500;
- Select records from a table and insert into another table.

+ Import/Execute queries from a .sql file:
- SOURCE '/path/of/sql/file/on/client/machine';

+ Import records from a comma seperated file from client:
- CREATE TABLE PEOPLE (FNAME VARCHAR(30), LNAME VARCHAR(30), GENDER ENUM('Male', 'Female'), BIRTH DATE);
- LOAD DATA LOCAL INFILE 'people.csv'
INTO TABLE PEOPLE
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
(FNAME, LNAME, GENDER, BIRTH);
- Insert records from a comma separated file on client machine (LOCAL) into the table.
- Default fields terminated by '\t'.
- Default lines terminated by '\n'.
- Columns must be specified if want to load only selected columns in table from file.
- Additional clauses:
ENCLOSED BY '"' (after FIELDS TERMINATED BY) - if fields are enclosed in special chars.
ESCAPED BY '\\' (after FIELDS TERMINATED BY) - if fields contains comma, they should \,
IGNORE 1 LINES - to ignore header line from data file.

+ DQL - SELECT query:
- SELECT * FROM BOOKS;
- Select all columns from the table.
- SELECT ID, NAME, AUTHOR, SUBJECT, PRICE FROM BOOKS;
- Select all columns (specify each column) from the table.
- SELECT ID, NAME FROM BOOKS;
- Select particular columns from the table.

+ Arithmetic operators:
- (), /, *, +, - : All these operator have precedance in given order.
- SELECT ID, NAME, PRICE + PRICE * 0.10 FROM BOOKS;
    - Select table columns and computed columns/derived columns/virtual columns from table. "PRICE + PRICE * 0.10" is computed column.
- SELECT ID AS BOOKNO, NAME, PRICE + PRICE * 0.05 AS FINAL_PRICE FROM BOOKS;
    - Alias can be used for the columns and computed columns using ANSI keyword "AS".
- SELECT ID AS BOOKNO, NAME, PRICE + PRICE * 0.05 AS "FINAL PRICE" FROM BOOKS;
    - Multi-word alias should be enclosed in double-quotes or back-quotes.
- SELECT ID BOOKNO, NAME, PRICE + PRICE * 0.05 `FINAL PRICE` FROM BOOKS;
    - AS keyword is optional.

+ DISTINCT columns:
- To retrieve unique column values from database.
- SELECT DISTINCT AUTHOR FROM BOOKS;
    - Fetch unique authors from table.
- SELECT DISTINCT SUBJECT, AUTHOR FROM BOOKS;
    - Fetch unique combination of subject and authors from table.

+ SELECT -- LIMIT clause:
- SELECT ID, NAME, AUTHOR, SUBJECT, PRICE FROM BOOKS LIMIT 5;
    - Fetch 5 rows from database table.
- SELECT ID, NAME, AUTHOR, SUBJECT, PRICE FROM BOOKS LIMIT 3, 5;
    - Skip first 3 rows and fetch 5 rows from table.

+ SELECT -- ORDER BY clause:
- In DBMS, records are stored sequentially into file.
- In RDBMS, data is not stored sequentially. In RDBMS table is not a file, rather each row is a file. So all rows are scattered (fragmented) in DB server hard disk.
- This scattered arrangement of rows, ensure better performance for INSERT operation.
- Since database is accessed by multiple users simultaneously, the multiple INSERT operations add the records in database as per free space available in DB server disk.
- Also in UPDATE operation if record size is increased beyond available free space after that record, the record will be moved on some new address on disk.
- For these reasons, SELECT operation does not guarantee the order of rows in output.

- SELECT ID, NAME, AUTHOR, SUBJECT, PRICE FROM BOOKS ORDER BY NAME ASC;
    - Fetch rows in ascending order of name. The "ASC" keyword after NAME is optional, because ascending is default sort order.
- SELECT ID, NAME, AUTHOR, SUBJECT, PRICE FROM BOOKS ORDER BY PRICE DESC;
    - Fetch rows in descending order of price.
- SELECT ID, NAME, AUTHOR, SUBJECT, PRICE FROM BOOKS ORDER BY SUBJECT, AUTHOR;
    - Fetch rows in ascending order of subject and order. Note that SUBJECT is first sort order, while AUTHOR is second sort order.
- There is no limit on number of columns in ORDER BY clause. However since all the sorting operations are done in DB server RAM, more sorting columns will decrease the performance.
- SELECT ID, NAME, AUTHOR, SUBJECT, PRICE FROM BOOKS ORDER BY SUBJECT DESC, PRICE ASC;
    - Each column in ORDER BY can be specified ASC or DESC separately.

- SELECT ID, NAME, AUTHOR, SUBJECT, PRICE + PRICE * 0.05 AS FINAL FROM BOOKS ORDER BY FINAL;
    - In MySQL, computed columns can be used in ORDER BY clause. This is not supported in all databases.
- SELECT ID, NAME, AUTHOR, SUBJECT, PRICE + PRICE * 0.05 AS FINAL FROM BOOKS ORDER BY 5;
    - The ORDER BY can mention the column number in SELECT query. This is supported in all RDBMS.

+ SELECT -- with Criteria -- WHERE clause.
- SELECT * FROM BOOKS WHERE SUBJECT='C';
- SELECT * FROM BOOKS WHERE AUTHOR='SCHILDT';
- SELECT * FROM BOOKS WHERE AUTHOR!='KANETKAR';
- SELECT * FROM BOOKS WHERE AUTHOR='SCHILDT' OR SUBJECT='JAVA';
- SELECT * FROM BOOKS WHERE AUTHOR='SCHILDT' AND SUBJECT='JAVA';

- WHERE clause -- Relational/Comparison operators:

=	Equal to
!= <>	Not Equal
>	Less than
<	Greater than
>=	Greater than Equal to
<=	Less than Equal to

- These operators cannot be used to compare with "NULL" value. It is considered as false condition.
- In MySQL database, string data is case insensitive. So MySQL is more user-friendly.
- All these operators can be used for numeric as well as string operations.
- SELECT NAME FROM BOOKS WHERE NAME >'D';
    - In MySQL, when two strings of different length are compared, then the shorter string is temporarily padded with blank spaces on right side to make both strings are of same length. Then strings are compared on ASCII values.

- WHERE clause -- BETWEEN operator:
- SELECT * FROM BOOKS WHERE PRICE BETWEEN 300.00 AND 600.00;
- SELECT * FROM BOOKS WHERE NAME BETWEEN 'D' AND 'P';
    - Note that for the BETWEEN operator, the first value is included or second value is excluded.
- SELECT * FROM BOOKS WHERE PRICE NOT BETWEEN 300.00 AND 600.00;
    - NOT BETWEEN is inversion of the BETWEEN operator.
- The same SQL queries can be written using logical operator AND (by combining two conditions i.e. PRICE > 300.00 AND PRICE < 600.00). However BETWEEN is precompiled operator for RDBMS i.e. its execution plan is ready in database engine, so BETWEEN is faster in execution.

- WHERE clause -- IN operator:
- SELECT * FROM BOOKS WHERE ID IN (2001, 2003, 3002, 3003);
- SELECT * FROM BOOKS WHERE AUTHOR IN ('Yashwant Kanetkar', 'Herbert Schildt');
    - The same SQL queries can be written using logical operator OR (i.e. ID=2001 OR ID=2003 OR ID=3002 OR ID=3003). However IN is precompiled operator for RDBMS i.e. its execution plan is ready in database engine, so IN is faster in execution.
- SELECT * FROM BOOKS WHERE AUTHOR NOT IN ('Yashwant Kanetkar', 'Herbert Schildt');
    - NOT IN is inversion of IN operator.

- WHERE clause -- LIKE operator:
- Used to match the string pattern.
- It supports two wild card characters:

% : any number of any characters. e.g. 'Java%', '%Lang', '%an%'
_ : any one character. e.g. '____Complete Reference'

- SELECT * FROM BOOKS WHERE NAME LIKE 'Java%';
- SELECT * FROM BOOKS WHERE NAME LIKE '%Language';
- SELECT * FROM BOOKS WHERE NAME LIKE '%an%';
- SELECT * FROM BOOKS WHERE NAME LIKE '____Complete Reference';
- SELECT * FROM BOOKS WHERE NAME NOT LIKE '%an%';
- NOT LIKE is inversion of LIKE operator.

- WHERE clause -- Logical operators: AND, OR, NOT
- SELECT * FROM BOOKS WHERE SUBJECT='OS' OR AUTHOR='Schildt' AND PRICE > 500;
- SELECT * FROM BOOKS WHERE SUBJECT='OS' OR (AUTHOR='Schildt' AND PRICE > 500);
    - Both above statements are similar because, AND have higher precedence than OR.
- SELECT * FROM BOOKS WHERE (SUBJECT='OS' OR AUTHOR='Schildt') AND PRICE > 500;
    - This query produces different results than above queries.
- SELECT * FROM BOOKS WHERE NOT SUBJECT='OS' OR AUTHOR='Schildt' AND PRICE > 500;
- SELECT * FROM BOOKS WHERE NOT (SUBJECT='OS') OR AUTHOR='Schildt' AND PRICE > 500;
    - Both above statements are similar because, NOT have higher precedence than AND & OR.

- SELECT CASE statement:

```
- SELECT
CASE
WHEN ID=1001 THEN 'First'
WHEN ID=1002 THEN 'Second'
WHEN ID=1003 THEN 'Third'
ELSE 'Other'
END
FROM BOOKS;

- SELECT PRICE,
CASE
WHEN SIGN(PRICE - 500.0)=1 THEN 'PRICE + PRICE * 0.15'
WHEN SIGN(PRICE - 500.0)=0 THEN 'PRICE + PRICE * 0.10'
WHEN SIGN(PRICE - 500.0)=-1 THEN 'PRICE + PRICE * 0.05'
END
FROM BOOKS;
```

+ Single Row Functions:
- String Functions:
    - LENGTH(...), CONCAT(...), UPPER(...), LOWER(...), LEFT(), RIGHT()
    - SUBSTR(COL, START) --> FROM GIVEN INDEX TILL END. START CAN BE +VE (FROM START INDEX) OR -VE (FROM END INDEX).
    - SUBSTR(COL, START, LEN) --> FROM GIVEN INDEX NEXT LEN CHARS. LEN CAN BE +VE.
    - LPAD(...), RPAD(...), LTRIM(...), RTRIM(...), TRIM(...), REPLACE(...), REVERSE(...), INSTR(...), ASCII(...), CHAR(...)

- Numeric Functions:
    - ROUND(col), ROUND(col,2), ROUND(col,-2).
    - TRUNCATE(col,2), TRUNCATE(col, -2),
    - FLOOR(col), CEIL(col)

- SIGN(), MOD(num,den)
- SQRT() -- only for +ve values
- POWER(), ABS()
- LOG(n,m) -- LOG(10, 100);
- LN(n)
- SIN(x), COS(x), TAN(x) -- x in radians

- Date Functions:
  - Default format for DATE in DB is 'YYYY-MM-DD'.
  - SYSDATE() -- datetime of server when this function is executed on server.
  - NOW() -- datetime of server when execution of the query begun.
  - SLEEP() -- wait for number of seconds.
  - ADDDATE(date, days); --
  - DATE_ADD(date, INTERVAL 2 month)
  - DATEDIFF(date1, date2);
  - LAST_DAY()
  - DAYNAME(), MONTHNAME()
  - DAY(), MONTH(), YEAR()
  - ADDTIME(time, seconds);
  - TIME(), DATE()
  - DATE_FORMAT() -- %d, %D, %m, %b, %M, %y, %Y

- List Functions:
  GREATEST(val1, val2, ..., val255)
- SELECT GREATEST(PRICE, 200) FROM BOOKS;
- GREATEST(NUM1, NUM2, ...) -- returns greatest number.
- GREATEST(STR1, STR2, ...) -- returns greatest string.
- GREATEST(date1, date2, ...) -- returns greatest date.
- LEAST(val1, val2, ..., val255)
- SELECT LEAST(PRICE, 600) FROM BOOKS;
- LEAST(NUM1, NUM2, ...) -- returns least number.
- LEAST(STR1, STR2, ...) -- returns least string.
- LEAST(date1, date2, ...) -- returns least date.

+ DML -- UPDATE query:
- UPDATE BOOKS SET PRICE=871.212 WHERE ID=3002;
  - Update single column of single row with fixed value.
- UPDATE BOOKS SET PRICE=PRICE+PRICE*0.05 WHERE ID=3002;
  - Update single column of single row with computed value.
- UPDATE BOOKS SET PRICE=PRICE+PRICE*0.05, SUBJECT='Java' WHERE ID=3002;
  - Update multiple columns of single row.
- UPDATE BOOKS SET PRICE=PRICE+PRICE*0.05 WHERE SUBJECT='OS';
  - Update multiple rows.
- UPDATE BOOKS SET PRICE=PRICE+PRICE*0.05 WHERE SUBJECT='C' AND AUTHOR='Kanetkar';
  - Update multiple rows with logical operator in WHERE clause.
- UPDATE BOOKS SET PRICE=PRICE+PRICE*0.10;
  - Update all rows.
- This is DML query and hence can be rollbacked.
- Multiple rows and columns can be updated simultaneously; but only single table can be updated at a time. For multiple tables need multiple UPDATE commands.

+ DML -- DELETE query:

- DELETE FROM BOOKS WHERE ID=3002;
  - Delete record with given id.
- DELETE FROM BOOKS WHERE SUBJECT='Java';
  - Delete records of given subject.
- DELETE FROM BOOKS;
  - Delete all records.
- This is DML query and hence can be rollbacked.
- Table structure is not deleted.

+ DDL -- DROP query:
- TRUNCATE TABLE BOOKS;
- Delete all records. Cannot use WHERE clause.
- This is DDL query. So cannot be rollbacked.

+ DDL -- DROP query:
- To delete the whole table, use drop command.
- Only one table can be dropped at a time. Multiple DROP statements needed for deleteing multiple tables.
- DROP cannot be used with WHERE clause.

+ DTL / TCL (Transaction Control Language):
- Transaction is set of queries executed as a single unit. If any of the query from that unit fails, remaining queries effect will be reversed.
- By default, MySQL runs with auto-commit mode enabled. This means that as soon as you execute a statement that updates (modifies) a table, MySQL stores the update on disk to make it permanent. The change cannot be rolled back.
- To disable auto-commit mode implicitly for a series of statements, use the START TRANSACTION statement.
- With START TRANSACTION, auto-commit remains disabled until you end the transaction with COMMIT or ROLLBACK. The auto-commit mode then reverts to its previous state.
- The optional ANSI "WORK" keyword is supported for COMMIT and ROLLBACK.

- Transaction examples:

```
START TRANSACTION; // begin transaction.
... // one or more DML queries.
COMMIT; // save changes (from last committed state) into database.
-
START TRANSACTION; // begin transaction.
... // one or more DML queries.
ROLLBACK; // revert changes (to last committed state).
-
START TRANSACTION; // begin transaction.
... // one or more DML queries.
SAVEPOINT s1;
... // one or more DML queries.
SAVEPOINT s2;
... // one or more DML queries.
ROLLBACK TO s1; // revert changes state s1.
... // additional DML queries in transaction.
COMMIT; // or ROLLBACK – end of transaction.
```

- Commit will save all changes since last committed state.

```
COMMIT;
COMMIT WORK; // WORK is ANSI keyword and is optional.
```

- Rollback will undo all changes since last committed state.

```
ROLLBACK;
```

ROLLBACK WORK; // WORK is ANSI keyword and is optional.
- Only the DML commands are affected by ROLLBACK & COMMIT.
- DDL commands are auto committed.
START TRANSACTION;
INSERT ...;
INSERT ...;
INSERT ...;
CREATE TABLE ...; // all above commands committed.
ROLLBACK; // nothing to rollback.
- When exit from MySQL command line client (using EXIT), it will automatically commit. (Not observed).
- Any kind of power failure, system failure, window close, network failure or any other failure will rollback your last uncommitted transaction.
- SAVEPOINT is a flag. It represents a point within your work. You can rollback to specific savepoint.
- COMMIT will save all the DML changes since the last committed state. You cannot commit to specific savepoint.
- ROLLBACK or COMMIT will clear all intermediate savepoints.

+ Transaction support ACID properties:
- Atomic -- Multiple queries executed as SINGLE unit.
- Consistent -- The database should gain consistent state at the end of transaction.
- Isolated -- Concurrent transactions should be executed in isolation with each other.
- Durable -- Changes in DB should be persistent until next transaction is committed.

+ The SELECT operation from the table shows all data committed by other users and uncommitted data in current session/transaction. Example:

USER1 --> BOOKS TABLE	USER2 --> BOOKS TABLE
1. SELECT -> 100 rows	
	2. SELECT -> 100 rows
3. INSERT -> 10 rows (in transaction)	
	4. SELECT -> 100 rows
5. SELECT -> 110 rows	
6. COMMIT;	
	7. SELECT -> 110 rows

+ All operations from all users are sent to the server and maintained in a Request Queue. Then each operation from that queue is carried out one by one in FIFO manner.
+ Rows Locking:
+ Optimistic Locking:
- When user1 update or delete a row, that row is locked and becomes read-only for other users. If other users try to modify or delete such locked row, their transaction processing is blocked until row is unlocked.
- That row can be selected by other users, until user1 commit the changes.
- The other users can INSERT into that table.
- The other users can UPDATE or DELETE the other rows into that table.
- The locks are automatically released when COMMIT/ROLLBACK is done by that user (i.e. user1).
- This whole process is done automatically in MySQL and is known as "OPTIMISTIC LOCKING".

+ Pessimistic Locking:
- Manually locking the row in advanced before issuing UPDATE or DELETE is known as "PESSIMISTIC LOCKING".
- SELECT * FROM BOOKS WHERE SUBJECT='C' FOR UPDATE;
- FOR UPDATE should be the last clause of SELECT. Now all selected rows are locked, until transaction is committed or rollbacked.
- If these rows are already locked by another users, the SELECT operation is blocked until rows lock is released.

+ NULL values:

- INSERT INTO BOOKS(ID, NAME) VALUES(5001, 'The Alchemist');
    - remaining columns will be set to NULL.
- INSERT INTO BOOKS(ID, NAME, AUTHOR, SUBJECT, PRICE) VALUES(5002, 'Fountain Head', NULL, NULL, 631.319);
    - explicitly inserting NULL columns into the table.
- SELECT * FROM BOOKS WHERE SUBJECT=NULL;
    - does not select any column from table. NULL is compared using any operator.
- SELECT * FROM BOOKS WHERE SUBJECT<=>NULL;
    - fetch all rows where SUBJECT is null. <=> operator can compare NULL values.
- SELECT * FROM BOOKS WHERE SUBJECT IS NULL;
    - fetch all rows where SUBJECT is null.
- SELECT * FROM BOOKS WHERE SUBJECT IS NOT NULL;
    - fetch all rows where SUBJECT is not null.
- SELECT PRICE + PRICE * 0.05 FROM BOOKS;
    - returns null for the row containing null price.
- SELECT IFNULL(PRICE, 0) FROM BOOKS;
    - returns 0 if price is null; otherwise return price.
- If a row contains few columns as null values, some marker is stored there to indicate the null. However is last column(s) of row contains null, then nothing is stored there and hence disk space is saved. If certain columns of table are expected to have lot of null values in records, then it is recommended to keep such columns at the end in table schema/structure to save the disk space.

+ Group/Aggregate functions:
- These functions process data from multiple rows, hence also called as "multi-row" functions.
- The NULL values are not considered by the group functions. Use IFNULL() at appropriate places like AVG(), COUNT(), ...
- Examples:
- SELECT SUM(PRICE) FROM BOOKS;
- SELECT AVG(PRICE) FROM BOOKS;
- SELECT MIN(PRICE), MAX(PRICE) FROM BOOKS;
- SELECT COUNT(PRICE) FROM BOOKS;
- SELECT STDDEV(PRICE) FROM BOOKS;
- SELECT VARIANCE(PRICE) FROM BOOKS;
- SELECT SUBJECT, COUNT(IFNULL(PRICE,0)), SUM(PRICE), AVG(IFNULL(PRICE,0)), MIN(IFNULL(PRICE,0)), MAX(IFNULL(PRICE,0)) FROM BOOKS WHERE SUBJECT='C';
    - print summary report of books of subject 'C'.
- Restrictions on Group functions:

SELECT NAME, SUM(PRICE) FROM BOOKS; // Error.
    - Cannot select a regular column along with group function.

SELECT UPPER(NAME), SUM(PRICE) FROM BOOKS; // Error.
    - Cannot select single row function along with group function.

SELECT * FROM BOOKS WHERE PRICE > AVG(PRICE); // Error.
    - Cannot use group function in WHERE clause;

SELECT MAX(SUM(PRICE)) FROM BOOKS; // Error
    - Cannot nest group functions -- works in Oracle.

+ SELECT -- GROUP BY clause:
- Used to group similar values together. Helpful in getting summary of data.
- SELECT SUBJECT, COUNT(PRICE), SUM(PRICE), AVG(PRICE), MIN(PRICE), MAX(PRICE) FROM BOOKS GROUP BY SUBJECT;
    - generates subject-wise summary report -- matrix report.
    - When above query is executed, following steps are done in database server (execution plan):

a. Fetch all rows into database server RAM. -- only SUBJECT & PRICE columns.
b. Sort rows on SUBJECT.
c. Group all similar SUBJECT together.
d. Execute aggregate functions on group of records.

- Rules for GROUP BY:

- Each column in SELECT must be present in GROUP BY.
SELECT AUTHOR, SUM(PRICE) FROM BOOKS; // X
SELECT AUTHOR, SUM(PRICE) FROM BOOKS GROUP BY SUBJECT;// X
SELECT SUBEJCT, SUM(PRICE) FROM BOOKS GROUP BY SUBJECT;//√
- Each column in GROUP BY may or may not be present in SELECT.
SELECT SUM(PRICE) FROM BOOKS GROUP BY SUBJECT; // √
SELECT SUBJECT, SUM(PRICE) FROM BOOKS GROUP BY SUBJECT; // √
- There is no upper limit on number of columns in GROUP BY clause.
- Each GROUP BY query is known as "Two dimensional query"; because one can plot a graph from the generated output.
- If having two columns in group by, then it is 3-D query.
- If having three columns in group by, then it is 4-D query.
- Multi-dimensional queries are known as "Spatial queries".
- If you have a large number of columns in GROUP BY clause, it will be slower; because internally all those records need to be sorted.
- The order of columns in SELECT and GROUP BY need not to match.
- The order of columns in SELECT determines the position of column in output.
- The order of columns in GROUP BY determine sorting order, grouping order and aggregate operation order and hence affect speed of processing.
SELECT SUBJECT, AUTHOR, SUM(PRICE) FROM BOOKS GROUP BY SUBJECT, AUTHOR;
SELECT SUBJECT, AUTHOR, SUM(PRICE) FROM BOOKS GROUP BY AUTHOR, SUBJECT;

+ SELECT -- GROUP BY -- HAVING clause:
- SELECT SUBJECT, SUM(PRICE) FROM BOOKS GROUP BY SUBJECT HAVING SUM(PRICE) > 1000; // √
- SELECT SUBJECT, SUM(PRICE) FROM BOOKS GROUP BY SUBJECT HAVING PRICE > 1000; // X
    - HAVING clause can only contain aggregate functions or columns in GROUP BY.
- SELECT SUBJECT, SUM(PRICE) FROM BOOKS GROUP BY SUBJECT HAVING SUBJECT!='C'; // √
    - Even though this is valid, it is not recommended.
    - HAVING clause should have only aggregate functions -- better performance.
- SELECT SUBJECT, SUM(PRICE) FROM BOOKS WHERE SUBJECT!='C' GROUP BY SUBJECT; // efficient
- SELECT SUBJECT, SUM(PRICE) FROM BOOKS GROUP BY SUBJECT HAVING SUM(PRICE) > 1000 AND SUM(PRICE) < 3000; // correct

+ SELECT Syntax:
SELECT ..... FROM
WHERE ....
GROUP BY ....
HAVING ....
ORDER BY ....
LIMIT .., ..;

+ Joins:
- Un-necessary duplication of data leads to wastage of disk space.
- Designing tables so that duplication is not done is known as "Normalisation".
- To combine/view the columns of two or more tables together, join queries are used.
- There are five types of joins:

1. Cross Join:
- Compares each row of one table with each row of another table.
- It gives all possible combinations of table1 and table2.
- SELECT * FROM EMP CROSS JOIN DEPT;
- SELECT * FROM EMP, DEPT; // same as above
    - This is join without any WHERE clause. It is also known as "Cartesian Join".

- In MySQL, The **larger** table in Join (i.e. EMP) is referred as "Driving Table", while **smaller** table in Join (i.e. DEPT) is referred as "Driven Table". Each row of Driving table is combined with every row of Driven table.
- Cross join is the fastest join, because there is no condition check involved. This join internally follows nested loop.

```
for(EMP e : empList) {
        for(DEPT d : deptList) {
        fetch(columns from DEPT & EMP);
        }
}
```

2. Inner Join:

- The inner JOIN is used to return rows from both tables that satisfy the given condition.
- SELECT * FROM EMP INNER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;
- SELECT * FROM EMP, DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO; // same as above.
    - Fetch rows from EMP table and display DEPT corresponding to them. If any employee is not assigned a dept or dept not having employees will be skipped.
    - If condition in ON clause is of equality, this join is also referred as "Equi-Join".
    - If condition in ON clause is of inequality, this join is also referred as "InEqui-Join".

```
for(EMP e : empList)  {
        for(DEPT d : deptList)  {
        if(e.DEPTNO == d.DEPTNO) // equi-join
        fetch(columns from DEPT & EMP);
        }
}
```

3. Left Outer Join:

- Outer JOINs return all records matching from both tables.
- Left outer join fetch all rows from left table and matching rows from right table. If no matching rows found in right table, NULL values are returned.
- SELECT EMP.ENAME, DEPT.DNAME FROM EMP LEFT OUTER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;
    - Fetch all rows from EMP table and display DEPT corresponding to them. If any employee is not assigned a dept, then dept name will be displayed as NULL.
    - The OUTER keyword is optional. This join is also known as "Left Join".
- SELECT EMP.ENAME, DEPT.DNAME FROM DEPT LEFT OUTER JOIN EMP ON EMP.DEPTNO = DEPT.DEPTNO;

```
for(DEPT d : deptList) {
        match = false;
        for(EMP e : empList) {
        if(e.DEPTNO == d.DEPTNO) {
        match = true;
        fetch(colmns from DEPT & EMP);
        }
        }
        if(!match) // match == false
        fetch(columns from DEPT & a null row from EMP);
}
```

4. Right Outer Join:

- Right outer join fetch all rows from right table and matching rows from left table. If no matching rows found in left table, NULL values are returned.

- SELECT EMP.ENAME, DEPT.DNAME FROM EMP RIGHT OUTER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;
    - Fetch all rows from DEPT table and display EMP corresponding to them. If any department is not having employees, then emp name will be displayed as NULL.
- The OUTER keyword is optional. This join is also known as "Right Join".
- SELECT EMP.ENAME, DEPT.DNAME FROM DEPT RIGHT OUTER JOIN EMP ON EMP.DEPTNO = DEPT.DEPTNO;

```
for(EMP e : empList) {
    match = false;
    for(DEPT d : deptList) {
        if(e.DEPTNO == d.DEPTNO) {
            match = true;
            fetch(colmns from DEPT & EMP);
        }
    }
    if(! match) // match == false
        fetch(colmns from EMP & a null row from DEPT);
}
```

* Full Outer Join is combination of Left Join & Right Join. It is supported in many databases, but not in MySQL. One can use UNION to get effect of Full Outer Join.

```
SELECT EMP.ENAME, DEPT.DNAME FROM EMP LEFT OUTER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO
UNION
SELECT EMP.ENAME, DEPT.DNAME FROM EMP RIGHT OUTER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;
```

5. Self Join

- When join is done on same table, then it is known as "Self Join". The both columns in condition belong to the same table.
- SELECT e1.ENAME AS EMP_NAME, e2.ENAME AS MGR_NAME
FROM EMP e1, EMP e2
WHERE e1.MGR = e1.EMPNO;
    - This is slowest join. This join is based on recursion.

- Joins can be specified on multiple tables.

[BOOKS]: ID (INT), NAME (CHAR), AUTHOR (INT), SUBJECT (INT), PRICE (DOUBLE)
[AUTHOR]: ID (INT), NAME (CHAR)
[SUBJECT]: ID (INT), NAME (CHAR)

```
SELECT B.ID, B.NAME, B.PRICE, A.NAME, S.NAME
FROM BOOKS B
INNER JOIN SUBJECT S
ON B.SUBJECT = S.ID
INNER JOIN AUTHOR A
ON B.AUTHOR = A.ID;
OR
SELECT B.ID, B.NAME, B.PRICE, A.NAME, S.NAME
FROM BOOKS B, SUBJECT S, AUTHOR A
WHERE B.SUBJECT = S.ID AND B.AUTHOR = A.ID;
```

+ Types of relationship:

1. One To One:

- [USER] ---------- [ADDRESS]
1 1
[USER] TABLE: USERID, NAME, PASSWORD, CONTACT,
[ADDRESS] TABLE: ADDRID, FLAT, BLDG, AREA, PIN, CITY, STATE

2. One To Many:
- [USER] ---------- [ORDERS]
1 *
[USER] TABLE: USERID, NAME, PASSWORD, CONTACT,
[ORDER] TABLE: ORDERID, DATE, STATUS, USERID

3. Many to One:
- [ORDERS] ---------- [USERS]
* 1
[USER] TABLE: USERID, NAME, PASSWORD, CONTACT,
[ORDER] TABLE: ORDERID, DATE, STATUS, USERID
4. Many to Many:
- [MEETINGS] ---------- [EMPLOYEES]
* *
[EMPMEETINGS]
[MEETINGS] TABLE: MEETINGID, DATE_TIME, TOPIC
[EMPLOYEES] TABLE: EMPNO, ENAME, SAL
[EMPEMEETINGS] TABLE: MEETINGID, EMPID

+ Sub-Queries:
- It is query within query. Typically it is SELECT within SELECT.
- Maximum nesting of 255 queries is supported in SQL.
- SELECT * FROM BOOKS WHERE PRICE = (SELECT MIN(PRICE) FROM BOOKS);
- Get the book of minimum price.
- This is single row sub-query i.e. sub-query returns single row result.
- Note that output of sub-query is input of main query.
- SELECT * FROM BOOKS WHERE PRICE > ALL(SELECT PRICE FROM BOOKS WHERE SUBJECT='C');
- Get books whose price is more than price of books of 'C'.
- This is multi-row sub-query i.e. sub-query returns multiple results.
- ALL operator compares with all values returned by sub-query. This does logical AND operation.
- SELECT DNAME FROM DEPT WHERE DEPTNO = ANY(SELECT DISTINCT DEPTNO FROM EMP);
- Display DNAME that contain employees.
- This is also multi-row sub-query. Note that the above sub-query processes all rows of the EMP table.
- ANY operator compares with any value returned by sub-query. This does logical OR operation similar to IN operator. However IN operator does only check equality, while ANY can be used for doing any comparison i.e. <, >, <=, >=.
- In MySQL, ANY & ALL operators can be used with sub-queries only.
- For each row of main query, sub-query is executed once. Hence sub-queries are slower than joins. Nesting multiple levels of sub-queries will reduce speed of execution. To improve the performance, try to reduce the number of rows returned by sub-query.
- SELECT D.DNAME FROM DEPT D WHERE DEPTNO IN (SELECT E.DEPTNO FROM EMP E WHERE E.DEPTNO = D.DEPTNO);
- For each row of outer table query, the sub-query fetch limited number of rows due to WHERE clause in sub-query. This is much efficient than usual sub-queries.

- This sub-query is using result of outer query in its WHERE clause; such sub-query is called as correlated sub-query. Correlated sub-queries execute faster than joins.
- SELECT D.DNAME FROM DEPT D WHERE EXISTS (SELECT E.DEPTNO FROM EMP E WHERE E.DEPTNO = D.DEPTNO);
    - EXISTS operator execute faster than IN, ANY or ALL clauses; because they just check if at least one row is returned by sub-query and doesn't perform any comparison.
- NOT EXISTS operator is negation of EXISTS operator.
- DELETE FROM BOOKS WHERE PRICE = (SELECT MAX(PRICE) FROM BOOKS); // Not supported in all DBs.
    - In MySQL, Sub-queries in UPDATE/DELETE is allowed, but sub-query should not SELECT from the same table on which UPDATE/DELETE is getting performed.

+ Indexes:
- CREATE INDEX SUBJECT_BOOKS_INDEX ON BOOKS (SUBJECT);
    - Faster searching is enabled on index columns.
- EXPLAIN FORMAT=JSON SELECT SUBJECT, SUM(PRICE) FROM BOOKS GROUP BY SUBJECT;

    - EXPLAIN select query can be used to see efficiency of the query. MySQL allows FORMAT=JSON for detailed description of execution of the query.
- One table can have multiple indices to improve search on multiple columns. *In MySQL maximum 64 indices are supported.*
- Creating index need additional space on server disk, because internally it creates some data structurw (like BTree or HashTable) on the disk.
- DML operations are slower when table has multiple indices, because for each DML operation indexes are updated.
- Similar values in index column are gathered together and stored as Btrees, which are used for any kind of comparison i.e. equality or >, <, ... For columns which needs only equality checks, index is stored as hash-table; that executes much faster.

CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));
    - To index TEXT/BLOB column, you must specify a prefix length for the index, in order keep size of index file minimum.
- NULL values are not stored in INDEX.

CREATE INDEX AUTHOR_BOOKS_INDEX ON BOOKS (AUTHOR DESC);
    - By default all indexes are in ascending order. But they can be created in desc order as well.

DROP INDEX AUTHOR_BOOKS_INDEX ON BOOKS;
    - Index can be dropped.
- When table is dropped, all its indexes are automatically dropped.

- Index types:
- Normal Index: Rows may or may not have duplicate values.
- Unique Index: Rows cannot have duplicate values.

CREATE UNIQUE INDEX index_name ON table_name (col_name);
- Composite Index: When lookup is based on multiple columns together. It improves searching if first index column or all index columns (or their single row functions) are used for search operation as logical AND.

CREATE TABLE PEOPLE (FNAME CHAR(20), LNAME CHAR(20), MOBILE CHAR(10), INDEX IDX_NAME(LNAME,FNAME));
- ... WHERE LNAME='A'; // √
- ... WHERE FNAME='X' AND LNAME='B'; // √
- ... WHERE FNAME='Y'; // X

- ... WHERE FNAME='Z' OR LNAME='C'; // X
- Indexes are used to improve performance for selecting rows with indexed columns (WHERE), ordered by indexed columns (ORDER BY), grouped by indexed columns (GROUP BY) and distinct values of index columns (DISTINCT).
- Primary key, unique columns and columns in join statements should be indexed.
- However using indexes for columns where searching/sorting on column is not needed frequently, indexing should be avoided.
- Finally indexes on a table can be displayed as
SHOW INDEXES FROM BOOKS;

+ Constraints:
- Constraints are limitations/restrictions imposed on table/columns.
1. PRIMARY KEY:
- Column or set of columns that uniquely identifies a row.
- Only one primary key is allowed.
- Duplicate values are not allowed for primary key.
- NULL values are not allowed for primary key.
- Search on primary key is faster (it is unique index, automatically created).
- It is recommended for each table to have primary key.
- TEXT & BLOB cannot be the primary key.
- If any column from table cannot be used as primary key, then some combination can be chosen as primary key (composite primary key) or some additional column can be used as primary key (surrogate key).
    - ALTER TABLE EMP DROP PRIMARY KEY;
    - ALTER TABLE EMP ADD PRIMARY KEY(EMPNO);

2. NOT NULL:
- NULL values are not allowed.
- duplicate values are allowed.

3. UNIQUE:
- Duplicate values are not allowed.
- NULL values are allowed.
- TEXT & BLOB cannot be unique.
- Search on unique key is faster (Internally unique index, automatically created).
- One or more unique key is allowed.

4. FOREIGN KEY:
- It is a column or set of columns that references a column of some table. If column belongs to the same table, it is "self referencing".
- Foreign key constraint is specified on child column (not on parent column).
- Foreign key can have duplicate values as well as null values.
- Cannot delete parent row, when child rows exists. However this can be done by providing "ON DELETE CASCADE" while making foreign key.
- Cannot update primary key from parent row, when child rows exists. However this can be done by providing "ON UPDATE CASCADE" while making foreign key.
- By default MySQL check the foreign key constraints. However these checks can be disabled by
    SET foreign_key_checks = 0;
-
CREATE TABLE DEPT (DEPTNO INT(4),

DNAME VARCHAR(50),
LOC VARCHAR(50),
CONSTRAINT PK_DEPT PRIMARY KEY(DEPTNO));
-
CREATE TABLE EMP (EMPNO INT(4),
ENAME VARCHAR(50),
SAL DOUBLE(7,2),
DEPTNO INT(4),
CONSTRAINT PK_EMP PRIMARY KEY(EMPNO),
CONSTRAINT FK_EMP_DEPT FOREIGN KEY(DEPTNO) REFERENCES DEPT(DEPTNO));
OR
CREATE TABLE DEPT (DEPTNO INT(4) PRIMARY KEY,
DNAME VARCHAR(50),
LOC VARCHAR(50));
-
CREATE TABLE EMP (EMPNO INT(4) PRIMARY KEY,
ENAME VARCHAR(50),
SAL DOUBLE(7,2),
DEPTNO INT(4) REFERENCES DEPT(DEPTNO)
);
-
DELETE FROM DEPT WHERE DEPTNO=10; // ERROR.
INSERT INTO EMP VALUES(1001, 'NILESH', 20000, 50); // ERROR.
UPDATE DEPT DEPTNO=60 WHERE DEPTNO=10; // ERROR.

OR

CREATE TABLE EMP (EMPNO INT(4),
ENAME VARCHAR(50),
SAL DOUBLE(7,2),
DEPTNO INT(4),
CONSTRAINT PK_EMP PRIMARY KEY(EMPNO),
CONSTRAINT FK_EMP_DEPT FOREIGN KEY(DEPTNO) REFERENCES DEPT(DEPTNO) ON DELETE CASCADE ON
UPDATE CASCADE);
-
DELETE FROM DEPT WHERE DEPTNO=10; // DELETE PARENT ROW WILL DELETE CORRESPONDING CHILD ROWS.
-
UPDATE DEPT DEPTNO=60 WHERE DEPTNO=10; // UPDATE PARENT ROW WILL UPDATE CHILD ROWS.
Constraints can be specified at two levels i.e. Column level and Table level.
- Column level constraints are specific to the column and mentioned immediately after the column definition.
CREATE TABLE EMP(
EMPNO INT PRIMARY KEY AUTO_INCREMENT,
ENAME VARCHAR(40) NOT NULL,
EMAIL VARCHAR(40) NOT NULL UNIQUE,
SAL FLOAT NOT NULL DEFAULT 0.0,
DEPTNO INT REFERENCES DEPT(DEPTNO)
);
- INSERT INTO EMP(ENAME, EMAIL, DEPTNO) VALUES('NILESH', 'NILESH@SUN.COM', 20);

- Table level constraints are not specific to a particular column and hence must be given at the end of CREATE TABLE statement e.g. composite PRIMARY KEY or composite UNIQUE key. All column level constraints can be given at table level except NOT NULL.
-

```
CREATE TABLE EMP(
EMPNO INT AUTO_INCREMENT,
FNAME CHAR(40) NOT NULL,
LNAME CHAR(40) NOT NULL,
EMAIL VARCHAR(40) NOT NULL,
SAL FLOAT NOT NULL DEFAULT 0.0,
DEPTNO INT,
PRIMARY KEY(EMPNO),
UNIQUE(FNAME, LNAME),
UNIQUE(EMAIL),
FOREIGN KEY(DEPTNO) REFERENCES DEPT(DEPTNO)
);
```

- Constraints are stored in MySQL system table information_schema.constraints.

+ DDL - ALTER statements:

- Direct changes:
- Rename a table.

```
RENAME TABLE PEOPLE TO CONTACTS;
```

- Add column to table.

```
ALTER TABLE CONTACTS ADD COLUMN EMAIL VARCHAR(30);
```

- Increase width of column.

```
ALTER TABLE CONTACTS MODIFY COLUMN EMAIL VARCHAR(40);
```

- Drop a column.

```
ALTER TABLE CONTACTS DROP COLUMN EMAIL;
```

- Indirect changes:
- Decrease width of column.

```
ALTER TABLE CONTACTS MODIFY COLUMN EMAIL VARCHAR(20);
```

In MySQL, the data in column will be truncated.

- Change data-type of column.

```
ALTER TABLE CONTACTS MODIFY COLUMN AGE INT(4);
```

In MySQL, if data in column is compatible, then change data type is done smoothly; otherwise data in column is lost.

- Copy table structure and data:

```
CREATE TABLE PEOPLE AS SELECT * FROM CONTACTS;
```

- Change order of columns:

```
CREATE TABLE PEOPLE AS SELECT LNAME, FNAME, BIRTH, GENDER FROM CONTACTS;
DROP TABLE CONTACTS;
RENAME TABLE CONTACTS TO PEOPLE;
```

- This is usually needed to save the disk space by shifting null valued columns at the end of table.
- The constraints can be added or removed using ALTER statement:
- To add/remove an auto-increment primary key:

```
ALTER TABLE test ADD id INT AUTO_INCREMENT PRIMARY KEY;
ALTER TABLE test DROP PRIMARY KEY;
```

- To add/remove unique key:

```
ALTER TABLE test ADD UNIQUE(colname);
```

ALTER TABLE test DROP INDEX(colnane);
- Changing next value of AUTO_INCREMENT in MySQL:
ALTER TABLE BOOKS AUTO_INCREMENT=500;
- Checking constraints:
- CHECK(expr) can be added at the end of CREATE TABLE statement or at column level for validations. Here expr contains column names along with checks using relational operators, logical operators, arithmetic operators and special operators like BETWEEN, IN, LIKE, etc. CHECK(expr) is ignored in MySQL 5 version.

```
CREATE TABLE EMP (
EMPNO INT PRIMARY KEY,
ENAME VARCHAR(20) CHECK (ENAME = UPPER(ENAME)),
SAL FLOAT CHECK (SAL BETWEEN 1000 AND 5000),
COMM FLOAT CHECK (COMM > 0),
CHECK (SAL+COMM < 10000)
);
```

+ Views:
- Facilitated by all RDBMSes and few DBMSes as well to provide restricted access to the users for security reasons.
- It provides indirect access to the table.
- Internally it stores address of table in form of hard disk pointer(address), known as "locator".
- Examples:

```
CREATE VIEW view1 AS (SELECT * FROM BOOKS);
CREATE VIEW view2 AS (SELECT ID,NAME,PRICE FROM BOOKS); // restricted columns
CREATE VIEW view3 AS (SELECT ID,NAME,PRICE+PRICE*0.05 FROM BOOKS); // computed columns
CREATE VIEW view4 AS (SELECT ID,NAME,PRICE FROM BOOKS WHERE PRICE > 500); // restricted rows
CREATE VIEW view5 AS (SELECT SUBJECT,SUM(PRICE) FROM BOOKS GROUP BY SUBJECT); // summary view
CREATE VIEW view6 AS (SELECT D.DEPTNO, D.DNAME, E.EMPNO, E.ENAME, E.SAL FROM DEPT D INNER JOIN EMP
E ON E.DEPTNO = D.DEPTNO); // joined view
```

- Views's data is not stored separately on disk, only SELECT statement is stored in system tables in compiled form.
- Since view statements are precompiled, they are quiet faster.
- DML operations can be performed on view, provided view doesn't contain computed columns or group by or any join and doesn't violate constraints on underlying table (like primary key or foreign key or not null).
- The DML operations performed on view impacts the underlying table.
- View can be created with CHECK OPTION to ensure that user don't perform invalid DML operations on view. It is like CHECK constraints.

```
CREATE VIEW view7 AS (SELECT ID, NAME, PRICE FROM BOOKS WHERE PRICE > 500) WITH CHECK OPTION;
INSERT INTO view7 VALUES (9001, 'ABC', 600); // correct
INSERT INTO view7 VALUES (9002, 'XYZ', 400); // wrong
```

- Views can be displayed:

```
SHOW TABLES; // show tables and views
SHOW FULL TABLES; // differentiate table and views
```

- View can be dropped:
DROP VIEW viewname;
- View can be edited:
CREATE OR REPLACE VIEW view2 AS (SELECT ID,NAME FROM BOOKS);
- Views can be created on top of another view:
CREATE OR REPLACE VIEW view8 AS (SELECT ID,NAME FROM view4);

This exceeds limit of SQL i.e. can have UNION of >255 SELECTS, >255 levels of sub-queries, ...

- Applications/Uses of views:

- Security purpose. Different users can be given different access and permissions by creating multiple views.
- Hides source code of table from user.
- Simplifies complicated queries.

+ Codd's Rules:

- Dr. Edgar F. Codd done extensive research in RDBMS and came up with 12 rules, which every RDBMS must follow.

Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

So databases have lot of tables for its manage its own information. Such databases in MySQL are "mysql", "information_schema" and "performance_schema". They store information like "mysql.users", "mysql.tables", ...

Rule 2: Guaranteed Access Rule

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following – data is missing, data is not known, or data is not applicable.

Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as data dictionary, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rule to apply.

Rule 10: Integrity Independence
A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

Rule 11: Distribution Independence
The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

Rule 12: Non-Subversion Rule
If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

+ MySQL Programming Language (PL):
- This is programming language for MySQL database, used for server-side processing of data.
- MySQL programs are executed in server RAM, when requested by client. The client can be MySQL command line client, MySQL workbench or any other program client. The client programs in Java for any RDBMS are written using JDBC (Java Database Connectivity) specification.
- MySQL programs are referred as PL blocks.
- It is block level language (like C) i.e. one block can be nested in another block.
- Advantages of block language are:
- Modularity (easier to manage larger programs).
- Encapsulation (Data hinding, Global & local variables).
- Error management (Exception of inner block can be handled in outer block).
- Console Input/output is not recommended in PL blocks.
- PL blocks contains data processing and transaction management logic.
- MySQL supports following SQL commands into PL block:
- INSERT, UPDATE, DELETE, CREATE, ALTER, DROP, COMMIT, ROLLBACK, SAVEPOINT, SELECT.
- DQL and DCL commands are not recommended; while DCL comamnds are not allowed in PL block.
- Typically output of PL blocks is stored in a seperate table, from where it can be accessed by others.

```
CREATE TABLE RESULTS(FIRST INT, SECOND VARCHAR(50), MODIFIED TIMESTAMP);
```

+ Session variables:
- Scope of session variables is limited to the client session.
- The session variable start with @ sign. It doesn't need a declaration. One can start using it directly setting its value.
- Session variable can be used in the queries and mainly while calling stored procedures.
- Example:

```
SET @ROWNUM=0;
SELECT @ROWNUM FROM DUAL;
SELECT @ROWNUM:=@ROWNUM+1 AS SRNO, ID, NAME, PRICE FROM BOOKS;
```

+ MySQL stored procedure:

- Stored Procedures are MySQL programs. They are stored on server side as objects in compiled format and hence executes faster than simple SQL queries.
- Procedure is set of SQL queries. It also allows statements like local variable declarations, if-else, case, loops, etc. One procedure can invoke another procedure. The stored procedures can be recursive.
- Procedure do not have return type i.e. return type is void.
- Procedures can be have one or more parameters, however stored procedures cannot be overloaded.
- Procedure entire code is stored in system table.

```
SHOW PROCEDURE STATUS;
```

- Stored procedures can be called using CALL keyword:

```
CALL PROC_NAME(...);
```

- Display procedure code:

```
SHOW CREATE PROCEDURE PROC_NAME;
```

- Procedures are executed in the server RAM when requested from client. These are global procedures and can be called from any client like MySQL prompt, MySQL workbench or any program (provided they have appropriate permissions).
- In case of multi-user environment, single copy of stored procedure is loaded into server RAM for multiple user requests. For a single threaded server, stored procedures are executed in FIFO manner; otherwise it follows time-sharing.
- Procedures can have three types of parameters:

a. IN parameter:
- Used to give input to the stored procedure. This is default mode.
- This is passed by value. Any changes in param within procedure are not reflected into original value.
- Can pass constant, expression or variable as parameter.

b. OUT parameter:
- Used to get the output from the stored procedure.
- This is passed by address. Changes in the variable within procedure are available in original variable.
- Can pass only variable.

c. INOUT parameter:
- Used to give input as well as get output from the stored procedure.
- This is passed by address. Its value can be processed in procedure and changes in the variable within procedure are available in original variable.
- Can pass only variable.
- Example:

```
CREATE PROCEDURE TEST13(IN P_IN INT, OUT P_OUT INT, INOUT P_INOUT INT)
BEGIN
-- CALC SQUARE OF P_IN & STORE IN P_OUT
SET P_OUT = P_IN * P_IN;
-- CALC CUBE OF P_INOUT & STORE IN P_INOUT
SET P_INOUT = P_INOUT * P_INOUT * P_INOUT;
END;

-- SET @A = 10;
-- SET @B = 0;
-- SET @C = 4;
-- CALL(A, B, C);
-- SELECT @A, @B, @C FROM DUAL;
```

+ MySQL Stored Functions:

- Stored Functions are MySQL programs. They are stored on server side as objects in compiled format and hence executes faster than simple SQL queries.
- Functions can be having one or more parameters. All params are by deafult IN. OUT or INOUT params are not allowed.
- Functions have return value and they must return some value using RETURN statement. Function must have one/more RETURN statements. When RETURN statement is executed control returns back to the caller, and further code from function is not executed.
- Function entire code is stored in system table.

```
SHOW FUNCTION STATUS;
SHOW CREATE FUNCTION FUNC_NAME;
```

- Since functions returns value, they are called from SELECT statement:

```
SELECT FUNC_NAME(...) FROM DUAL;
```

- Like procedures, functions allows statements like local variable declarations, if-else, case, loops, etc. One function can invoke another function/procedure and vice-versa. The functions can be recursive.
- There are two types of functions:

a. DETERMINISTIC:

- For same input parameters function always produces same results.
- MySQL may cache results of such functions (for given parameters) to speed-up the performance.

b. NOT DETERMINISTIC:

- For same input parameters function produces same/different results in each call.
- Usually these functions produces results based on current system date.
- MySQL do not try to optimize these functions.
- Setting up wrong type during function declaration may produce unexpected results and degrade the performance.
- Example:

```
CREATE FUNCTION TEST(p_name VARCHAR(20))
RETURNS VARCHAR(30)
DETERMINISTIC
BEGIN
        DECLARE v_name VARCHAR(20);
        SET v_name = UPPER(p_name);
        RETURN CONCAT('Hello ', v_name);
END;
```

- Example Call:

```
SELECT TEST('Nilesh') FROM DUAL;
SELECT TEST(FNAME) FROM CONTACTS;
```

- Even though functions can have SQL statements in it; it is recommended not to use them for sake of efficiency. Functions are executed for each row fetched from table and hence they should be light-weight as possible.

+ MySQL Error Handling:

- It is important to handle error in procedures/functions. One can choose to continue the procedure execution further (after handling the error) or exit from the procedure when error occurs.
- MySQL error handlers are declared as:

```
DECLARE action HANDLER FOR condition handler_impl;
```

- Here action can be:

a. CONTINUE: Continue the execution of the block in which error occured after completing the handler_impl.

b. EXIT: Exit the block in which error occured after completing the handler_impl.

- Here condition can be:

a. MySQL error code:
1062: Duplicate entry.
1054: Unknown column.
b. SQLSTATE value:
23000: Duplicate entry.
SQLWARNING: Shorthand for SQL warning.
SQLEXCEPTION: Shorthand for SQL error.
NOTFOUND: Shorthand for condition used for a CURSOR or SELECT INTO.
c. A named condition:
- DECLA RE duplicate_entry CONDITION FOR 1062;

- Here handle_impl can be a single expression or a block (BEGIN ... END).
- Example:

```
CREATE PROCEDURE SP_ADD_BOOK(IN P_ID INT, IN P_NAME VARCHAR(50))
BEGIN
        DECLARE EXIT HANDLER FOR 1062
        BEGIN
        INSERT INTO RESULTS(FIRST,SECOND) VALUES(P_ID, 'Already Present');
        END;

        INSERT INTO BOOKS(ID,NAME) VALUES(P_ID,P_NAME);
        -- ... will be skipped in case of duplicate primary key.
END;
```

- Example:

```
        CREATE PROCEDURE SP_ADD_BOOK(IN P_ID INT, IN P_NAME VARCHAR(50))
        BEGIN
        DECLARE EXIT HANDLER FOR 1062 SELECT 'Duplicate BookIds Not Allowed';
        INSERT INTO BOOKS(ID,NAME) VALUES(P_ID,P_NAME);
        -- ... will be skipped in case of duplicate primary key.
        END;
```

- Example:

```
        CREATE PROCEDURE SP_ADD_BOOK(IN P_ID INT, IN P_NAME VARCHAR(50))
        BEGIN
        DECLARE error_flag INT DEFAULT 0;
        DECLARE CONTINUE HANDLER FOR 1062 SET error_flag=1;
        INSERT INTO BOOKS(ID,NAME) VALUES(P_ID,P_NAME);
        -- ... will be executed irrespective of duplicate primary key.
        END;
```

+ MySQL Cursor:

- Cursor is a special variable used in procedures/functions to iterate/process a resultset row by row. Cursors are based on SELECT statement and fetch one row at a time, which may contain one or more columns (like 2-D array).
- MySQL cursors are read-only, non-scrollable and asensitive.
- Read-only: Cursor is used only to fetch the records (not to update or delete).
- Non-scrollable: Cursor iterate through the records in the order specified by SELECT statement. Reverse access or random access is not allowed.
- Asensitive: Cursor points to the actual address of data (not a copy of it). While cursor iteration is in progress, if any other clients update/delete the data, then changes are immediately reflected into cursor. So it is recommended not to update the data, while accessing using cursor.
- Cursor should be opened before fetching rows from it and should be closed after iteration is over.

Example:

```
CREATE PROCEDURE TEST_CURSOR()
BEGIN
        DECLARE V_I INT DEFAULT 1;
        DECLARE V_ID INT;
        DECLARE V_NAME VARCHAR(50);
        DECLARE V_CUR CURSOR FOR SELECT ID, NAME FROM BOOKS;
        OPEN V_CUR;
        WHILE V_I <= 5 DO
        FETCH V_CUR INTO V_ID, V_NAME;
        INSERT INTO RESULTS(FIRST, SECOND) VALUES(V_ID, V_NAME);
        SET V_I = V_I + 1;
        END WHILE;
        CLOSE V_CUR;
END;
```

- If cursor is not closed, it will be automatically closed at the end of block. For lengthy programs it is good to close cursor as soon as it is finished.
- If records are not fetched (reached at the end of result), the NOTFOUND error is set; which can be handled using error handler.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET V_DONE=TRUE;
```

- Cursor must be closed before reopening.

```
CLOSE V_CUR;
OPEN V  CUR;
```

- Note that cursor SELECT query may be simple or may also contains WHERE, GROUP BY, ORDER BY, LIMIT, sub-queries or it can be a join query. However syntax of iterating through cursor remains same.
- Multiple cursors can be opened simultaneously to fetch data from multiple tables and implement customized joins which may not straight-forward to implement in SQL.
- Also processing rows from cursor may include UPDATE, DELETE or INSERT operations on same/different table. Above example shows INSERT operation.
- If records of same table are to be updated, consider locking them while selecting using FOR UPDATE. This will ensure that other clients are not changing these records simultaneously.

+ Triggers:

- Triggers are MySQL programs. They are stored on server side as objects in compiled format and hence executes faster than simple SQL queries.
- Triggers are supported by all standard RDBMS like Oracle, MySQL, ...; they are not supported by WEAK RDBMS like MS-Access.
- Unlike Procedures & functions, triggers are never called explicitly by the client. As name suggests, their execution is triggered/caused by certain events. Triggers neither have parameters nor have return values.
- The possible events are.
- BEFORE INSERT, AFTER INSERT
- BEFORE UPDATE, AFTER UPDATE
- BEFORE DELETE, AFTER DELETE
- Triggers are applicable only for DML operations and supports pre and post processing.
- Trigger is a PL block, so it can contain all PL syntax like if-else, loop, case, cursors, etc. Also they can invoke any PROCEDURE or FUNCTION.
- Triggers are stored and executed on server side, when DML operations are done on table from any client (command line client, workbench or any programming client).

- COMMIT and ROLLBACK are not allowed within triggers. They are done in the transaction from which trigger is executed. Note that triggers are executed as a part of the DML operation transaction. Hence any failure in trigger will fail the whole transaction and thus DML operation will be rollbacked. This ensures data consistency.
- Example -- Executed for each row of BOOKS table after updating any column of BOOKS.

```
CREATE TRIGGER TEST_TRIG
AFTER UPDATE ON BOOKS
FOR EACH ROW
BEGIN
        INSERT INTO RESULTS (FIRST, SECOND) VALUES (OLD.PRICE, 'PRICE BEFORE UPDATE');
        INSERT INTO RESULTS (FIRST, SECOND) VALUES (NEW.PRICE, 'PRICE AFTER UPDATE');
END;
```

- NEW and OLD are MySQL extensions to triggers.
- INSERT triggers can access only NEW records.
- DELETE triggers can access only OLD records.
- UPDATE triggers can access NEW and OLD records.
- Applications/Use-cases:
    - Maintain logs of DML operations (Audit Trails).
    - Data cleansing before insert or update data into table. (Modify NEW value).
    - Copying each record AFTER INSERT into another table of the database, called as "Shadow table".
    - Data replication: Creating mirror copy of each row in table on some different server. In case of failure of database, standby server can be used.
    - Copying each record AFTER DELETE into another table of the database, called as "History table". This ensures that old data is not lost.
    - Auto operations of related tables using cascading triggers.
- Cascading Triggers:
    - One trigger causes second trigger to execute, second trigger causes execution of third trigger and so on.
    - In MySQL there is no upper limit on number of levels of cascading. This is helpful in complicated business processes.
- Mutating table error:
    - If cascading trigger causes one of the earlier trigger to re-execute, then this will not go in infinite loop. Rather database throws error "mutating table" and current transaction is rollbacked.
- To list triggers:
    - SHOW TRIGGERS;
    - SHOW TRIGGERS FROM db_name;
- To delete triggers;
    - DROP TRIGGER trigger_name;
- When a table is deleted, all its triggers are auto-deleted.

+ DCL -- GRANT & REVOKE:

- Security is built-in feature of any RDBMS. It is implemented in terms of permissions (a.k.a. privileges).
- Each entity created using CREATE statement is an object. E.g. TABLE, PROCEDURE, FUNCTION, TRIGGER, VIEW, INDEX, ...
- Two types of privileges:
- System privileges:
    - Privileges for certain commands i.e. CREATE TABLE, CREATE USER, CREATE TRIGGER, ...
    - Typically these privileges are given to the administrator.
    - For this each RDBMS has a default user. In MySQL, default user (at time of installation) is 'root'.
- Object privileges:
    - Privileges for certain objects i.e. TABLE, PROCEDURE, ...

- Can perform operations on the objects i.e. INSERT, UPDATE, DELETE, EXECUTE, ...
- Typically these privileges are given to the users.

- Examples -- System privileges:
GRANT CREATE TABLE TO dmc;
GRANT CREATE TABLE, CREATE PROCEDURE, CREATE FUNCTION TO dmc;
GRANT CREATE TABLE, CREATE PROCEDURE, CREATE FUNCTION TO dmc, dac;
- Examples -- Object privileges:
GRANT SELECT ON emp TO dmc;
GRANT SELECT,INSERT,UPDATE,DELETE ON emp TO dmc;
GRANT SELECT ON view1 TO dmc;
- Revoke:
- Used to remove the permissions/privileges.
REVOKE SELECT ON emp FROM dmc;
REVOKE UPDATE,DELETE ON emp FROM dmc;

+ MySQL Workbench:

- Desktop application for MySQL access/control.

+ phpmyadmin:

Is web-based software implemented in PHP to access/control MySQL database.
- Ubuntu install:
- sudo apt-get install apache2 php7.0 php7.0-mcrypt php7.0-mbstring phpmyadmin
- sudo vim /etc/apache2/apache2.conf
--> add line at the end :: Include /etc/phpmyadmin/apache.conf
- sudo systemctl restart apache2
- In browser: http://localhost/phpmyadmin

+ MonogoDb Installation -- Ubuntu:
sudo apt-get install mongodb
sudo apt-get install mongodb-server
sudo apt-get install mongodb-clients

+ MongoDb:
- sudo systemctl status mongodb
    - To check if mongodb is running on computer.
- sudo systemctl start mongodb
    - To start mongodb server
- sudo systemctl stop mongodb
    - To stop mongodb server
- sudo systemctl enable mongodb
    - To start mongodb server booting time.
- sudo systemctl disable mongodb
    - Not to start mongodb server booting time.
- netstat -tln
- 27017 : mongodb server port
- 28017 : mongodb server http interface port (if enabled in mongodb.conf)

- Mongo default data directory:
    - /var/lib/mongodb

- Mongo config file:
    - vim /etc/mongodb.conf

nohttpinterface = false
rest = true

- Mongo server process: mongod
    ps -e | grep "mongod"
- Mongo client shell -- follows JS queries: mongo

+ MongoDb:
- Developed by "10gen" systems -- Huge unstructured data handling capabilities.
- Database system: storage data, retrieval data, process data, query language (JS), partitioning, scaling out -- cluster management, fast searching (index), ...
- Made open-source by 10gen -- github.com -- main contributer mongodb.corp
- Each document is identified by a unique id -- "_id". If not given by client, then new ObjectId is created and added into document. This unique ObjectId is created by the client; it is of 12 bytes containing -- time in ms from epoch (timestamp), client machine id, counter,
- The _id for each document must be unique within collection.

+ MongoDb commands:
- show databases;
- use dbname;
- db;
- show collections;
- db.createCollection("colname");
- mongoimport -d test -c books --type csv --file ~/Documents/books.csv --headerline

```
- db.colname.insert({
"fname": "nilesh",
"lname": "ghule",
"mobile": "9527331338",
"email": "nilesh@sunbeaminfo.com",
});
- db.colname.insert({
fname: "nitin",
lname: "kudale",
mobile: "9881208115"
});
- db.colname.insert({
_id: "007",
fname: "james",
lname: "bond",
mobile: "9822012345"
});
- db.colname.find();
- displays first 20 records (on shell -- default limit).
- "it" command to see next 20.
- db.books.find().pretty();
- db.books.find({
}, {
_id:1,
name:1,
price:1
});
- db.books.find({
}, {
_id: 0,
name: 1,
author: 1,
price: 1
});
- db.books.find().limit(5);
- db.books.find().skip(3).limit(5);
- db.books.count();
- db.books.find().sort({
subject: 1
});
- db.books.find().sort({
subject: 1,
price: -1
});
- db.books.find({
subject: "C Programming"
});
- db.books.find({
subject: "C Programming",
```

```
author: "Yashwant Kanetkar"
});
- db.books.find({
$and : [
{ subject: "C Programming" },
{ author: "Yashwant Kanetkar" }
]
});
- db.books.find({
$or : [
{ subject: "C Programming" },
{ author: "Herbert Schildt" }
]
});
- db.books.find({
$nor : [
{ subject: "C Programming" },
{ author: "Herbert Schildt" }
]
});
- db.books.find({
price: { $gt : 500 }
});
$gt, $lt, $gte, $lte, $ne

- db.books.find({
price: { $gt : 400, $lt : 800 }
});
- db.books.find({
_id : { $in : [ 1001, 2001, 3001, 4001 ] }
});
$nin, $all
- db.books.aggregate({
$group : { "_id" : "$subject",
"total" : { $sum : "$price" },
"average" : { $avg : "$price" }
}
});
- db.colname.remove({_id : "007"});
deleteOne(), deleteMany()
- db.colname.update({
fname: "nitin"
}, {
mobile: "9881208114"
});
- db.colname.update({
fname: "nilesh"
}, {
$set : {
```

email: "ghule.nilesh@gmail.com"
}
});
- db.dept.insert({
_id: 10,
dname: "Accounts",
loc: "New York",
empList: [
{
ename: "Smith",
sal: 2324
},
{
ename: "Clerk",
sal: 2732,
comm: 345
}
]
});
- Relations are maintained using embedded objects.
- db.stud.insert({
_id: 100,
name: "rahul",
marks: 98.20,
addr: {
city: "karad",
dist: "satara"
}
});
- db.stud.find({
"addr.city": "karad"
});
- db.colname.drop();
- The update, delete operations are performed atomically on each document. The document in locked during update/delete, so that no other client can update/delete the document concurrently.

+ NoSQL Definition
- Database management system.
- No definition.
- Each NoSQL database is unique.

+ Characteristics of NoSQL:
- OpenSource (Mostly)
- Distributed -- Cluster Friendly
- High traffic (Designed for web)
- Schema-less
- Non-relational

+ Objectives of NoSQL:

- Scalability (Horizontal)
- Performance
- High Availability

+ RDBMS vs NoSQL:

SR	RDBMS	NoSQL
1.	More functionalities (Triggers, Cursors, Joins, Indexes, Views, ...)	Less functionalities
2.	Less performance	More performance
3.	Structured data (Tables/Columns)	Structured/Un-structured data

+ Cons of NoSQL:
- No joins support
- No complex transaction support
- No constraints support

+ Pros of NoSQL:
- Query Language
- Fast performance
- Horizontal scalability

+ When to use NoSQL?
- Ability to store and retrieve great quantity of data is needed.
- Relationship between entities is not important.
- Database is growing rapidly (twitter, facebook, blogs, ...).
- Data is un-strucured or changing with time.
- Fast application development with changing database scheme (agile development).
- Constraints/validation logic is needed not to be implemented in dbms.

+ When not to use NoSQL?
- Complex transactions to be handled.
- Joins to be handled by the dbms.
- Validations to be handled by dbms.