JAVA DAVE ENVIRONMENT

Advanced SQL, Ext JS, JSP & Servlet;

Java Boot Camp, August, 2017.

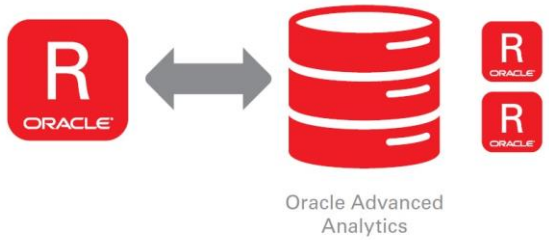
Dr. Kishore Biswas (Forrest/柯修)

PhD. Artificial Intelligence—Natural Language Processing.

CanWay IT Training ®.

CONTENTS

1) Advance SQL
2) JSP & Servlet
   - Introduction to JSP
   - JSP Life Cycle
   - Basic JSP Elements
   - Interacting with other Resources
3) Spring MVC
4) SQL practice



1) ADVANCE SQL

SQL JOIN

There are different types of joins available in SQL −

- INNER JOIN − returns rows when there is a match in both tables.
- LEFT JOIN − returns all rows from the left table, even if there are no matches in the right table.

- RIGHT JOIN − returns all rows from the right table, even if there are no matches in the left table.
- FULL JOIN − returns rows when there is a match in one of the tables.

- SELF JOIN − is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- CARTESIAN JOIN − returns the Cartesian product of the sets of records from the two or more joined tables.

SQL - INNER JOINS

- The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**.
- The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

SQL - INNER JOINS

The basic syntax of the INNER JOIN:

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

SQL - INNER JOINS

- The most important and frequently used of the joins is the INNER JOIN. They are also referred to as an EQUIJOIN.

- The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

UNION CLAUSE...

The SQL UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use this UNION clause, each SELECT statement must have

- The same number of columns selected
- The same number of column expressions
- The same data type and
- Have them in the same order

UNION CLAUSE...

SELECT column1 [, column2]
FROM table1 [, table2]
[WHERE condition]

UNION

SELECT column1 [, column2]
FROM table1 [, table2]
[WHERE condition]

SQL - INDEXES

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table.

- An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data.
- Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.
- Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

INDEX SYNTAX

Single-Column Indexes: A single-column index is created based on only one table column.

CREATE INDEX index_name

ON table_name (column_name);

Unique Indexes : Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows.

CREATE UNIQUE INDEX index_name

on table_name (column_name);

SQL - ALTER TABLE COMMAND

The SQL ALTER TABLE command is used to add, delete or modify columns in an existing table. You should also use the ALTER TABLE command to add and drop various constraints on an existing table.

- ALTER TABLE table_name ADD column_name datatype;
- ALTER TABLE table_name DROP COLUMN column_name;

SQL - USING VIEWS

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

- A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

- Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

SQL - USING VIEWS

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

SQL - INJECTION

- Injection usually occurs when you ask a user for input, like their name and instead of a name they give you a SQL statement that you will unknowingly run on your database. Never trust user provided data, process this data only after validation; as a rule, this is done by Pattern Matching.

- For example: the name is restricted to the alphanumerical characters plus underscore and to a length between 8 and 20 characters (modify these rules as needed).

SQL - INJECTION

```
• if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches)) {
•    $result = mysql_query("SELECT * FROM CUSTOMERS
•       WHERE name = $matches[0]");
• } else {
•    echo "user name not accepted";
• }
```

SQL FUNCTIONS

- SQL Aggregate functions: return a single value calculated from values in a column
  - AVG() – Returns the average value
  - COUNT() – Returns the number of rows
  - FIRST() – Returns the first value
  - LAST() – Returns the last value
  - MAX() – Returns the largest value
  - MIN() – Returns the smallest value
  - SUM() – Returns the sum

SQL FUNCTIONS

- SQL Scalar functions: returns a single value based on the input value
  - UCASE() – Converts a field to upper case
  - LCASE() – Converts a field to lower case
  - MID() – Extracts characters from a text field
  - LEN() – Returns the length of a text field
  - ROUND() – Rounds a numeric field to the number of decimals specified
  - NOW() – Returns the current system date and time
  - FORMAT() – Formats how a field is to be displayed

CREATE TABLE

products Table

P_Id	ProductName	UnitPrice	UnitsInStock	UnitsOnOrder
1	Jarlsberg	10.45	16	15
2	Mascarpone	32.56	23	
3	Gorgonzola	15.67	9	20

```
CREATE TABLE products (
            P_Id                INT,
            ProductName         VARCHAR(25),
            UnitPrice           decimal,
            UnitsInStock         INT,
            UnitsOnOrder         INT,
            PRIMARY KEY (P_Id)
);

INSERT INTO products VALUES (1, 'Jarlsberg', 10.45, 16, 15);
INSERT INTO products VALUES (2, 'Mascarpone', 32.56, 23,NULL);
INSERT INTO products VALUES (3, 'Gorgonzola', 15.67, 9, 20);
```

SQL AVG() FUNCTION

- AVG() returns the average value of a numeric column

```
SELECT AVG(column_name) FROM table_name
```

```
SELECT AVG(UnitPrice) AS UnitAverage FROM Products
```

UnitAverage
19.6667

```
SELECT ProductName FROM products
WHERE UnitPrice>( SELECT AVG(UnitPrice) FROM Products)
```

ProductName
Mascarpone

SQL COUNT( ) FUNCTION

- COUNT() returns the number of rows that matches a specified criteria

```
SELECT COUNT(column_name) FROM table_name
```

```
SELECT COUNT(*) FROM Products
```

count(*)
3

```
SELECT COUNT(DISTINCT column_name) FROM table_name
```

```
SELECT COUNT(DISTINCT city) As CityName From persons
```

CityName
2

SQL MAX() FUNCTION

- MAX() returns the largest value of the selected column

```
SELECT MAX(column_name) FROM table_name
```

```
SELECT MAX(UnitPrice) AS LargestUnitPrice FROM Products
```

LargestUnitPrice
33

SQL SUM() FUNCTION

- SUM() returns the total sum of the selected column

```
SELECT SUM(column_name) FROM table_name
```

```
SELECT SUM(UnitPrice) AS TotalUnitPrice FROM Products
```

TotalUnitPrice
59

CREATE CUMORDERS TABLE

O_Id	OrderDate	OrderPrice	Customer
1	2008/11/12	1000	Hansen
2	2008/10/23	1600	Nilsen
3	2008/09/02	700	Hansen
4	2008/09/03	300	Hansen
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

```
CREATE TABLE CumOrders (
        O_Id            INT,
        OrderDate       Date,
        OrderPrice      INT,
        Customer        Varchar(30),
        PRIMARY KEY (O_Id)  );
INSERT INTO CumOrders VALUES (1, '2011-08-06', 1000, 'Hansen');
INSERT INTO CumOrders VALUES (2, '2011-08-07', 1600, 'Nilsen');
INSERT INTO CumOrders VALUES (3, '2011-08-08', 700, 'Hansen');
INSERT INTO CumOrders VALUES (4, '2011-08-09', 300, 'Hansen');
INSERT INTO CumOrders VALUES (5, '2011-08-10', 2000, 'Jensen');
INSERT INTO CumOrders VALUES (6, '2011-08-11', 100, 'Nilsen');
```

SQL GROUP BY STATEMENT

- Aggregate functions often need an added GROUP BY statement to group the result-set by one or more columns

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
```

```
SELECT Customer, SUM(OrderPrice) FROM CumOrders
GROUP BY Customer
```

Customer	SUM(OrderPrice)
Hansen	2000
Jensen	2000
Nilsen	1700

Customer	OrderDate	SUM(OrderPrice)
Hansen	2011-08-06	1700
Hansen	2011-08-09	300
Jensen	2011-08-10	2000
Nilsen	2011-08-06	1600
Nilsen	2011-08-11	100

```
SELECT Customer, OrderDate, SUM(OrderPrice) FROM CumOrders
GROUP BY Customer, OrderDate
```

SQL HAVING CLAUSE

- HAVING was added to SQL because the WHERE keyword could not be used wit the aggregate functions

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value
```

```
SELECT Customer, SUM(OrderPrice) FROM CumOrders
GROUP BY Customer
HAVING SUM(OrderPrice)<2000
```

Customer	SUM(OrderPrice)
Nilsen	1700

SQL HAVING CLAUSE

```
SELECT Customer,  SUM(OrderPrice) FROM CumOrders
WHERE Customer='Hansen' OR Customer='Jensen'
GROUP BY Customer
HAVING SUM(OrderPrice)>1500
```

Customer	SUM(OrderPrice)
Hansen	2000
Jensen	2000

SQL UCASE() FUNCTION

- UCASE() converts the value of a field to uppercase.

```
SELECT UCASE(column_name) FROM table_name
```

```
SELECT UCASE(LastName) as LastName From Persons
```

LastName
HANSEN
SVENDSON
PETTERSEN
NILSEN
DING

SQL LCASE() FUNCTION

- LCASE() converts the value of a field to lowercase.

```
SELECT LCASE(column_name) FROM table_name
```

```
SELECT LCASE(LastName) as LastName From Persons
```

LastName
hansen
svendson
pettersen
nilsen
ding

SQL MID() FUNCTION

- MID() extracts characters from a text field

SmallCity
Sand
Sand
Stav
Stav
(NULL)

```
SELECT MID(column_name, start[, length]) FROM table_name
```

Parameter	Description
column_name	Required. The field to extract characters from
start	Required. Specifies the starting position (starts at 1)
length	Optional. The number of characters to return. If omitted, the MID() function returns the rest of the text

```
SELECT MID(City, 1, 4) as SmallCity From Persons
```

MySQL

```
SELECT substr(City, 1, 4) as SmallCity From Persons
```

PostgreSQL

SQL LENGTH() FUNCTION

- LENGTH() returns the length of the value in a text field.

```
SELECT LENGTH(column_name) FROM table_name
```

```
SELECT LENGTH(Address) as LengthOfAddress From Persons
```

LengthOfAddress
11
8
8
8
(NULL)

SQL ROUND() FUNCTION

- ROUND() rounds a numeric field to the number of decimals specified

```
SELECT ROUND(column_name, decimals) FROM table_name
```

Parameter	Description
column_name	Required. The field to round.
decimals	Required. Specifies the number of decimals to be returned.

```
UPDATE Products
SET UnitPrice=10.49
WHERE P_Id=1
```

```
SELECT ROUND(UnitPrice, 0) as Price From Products
```

Price
10
33
16

SQL DATE FUNCTIONS

- Introduces some build-in functions to deal with dates

MySQL Date Functions

Function	Description
NOW()	Returns the current date and time
CURDATE()	Returns the current date
CURTIME()	Returns the current time
DATE()	Extracts the date part of a date or date/time expression
EXTRACT()	Returns a single part of a date/time
DATE_ADD()	Adds a specified time interval to a date
DATE_SUB()	Subtracts a specified time interval from a date
DATEDIFF()	Returns the number of days between two dates
DATE_FORMAT()	Displays date/time data in different formats

MYSQL NOW() FUNCTION

- NOW() returns the current date and time

```
SELECT NOW(), CURDATE(), CURTIME()
```

MySQL

NOW()	CURDATE()	CURTIME()
2011-08-06 10:58:37	2011-08-06	10:58:37

```
SELECT NOW()
```

PostgreSQL

```
CREATE TABLE OrderCheese
(
OrderId int NOT NULL,
ProductName varchar(50) NOT NULL,
OrderDate datetime NOT NULL,
PRIMARY KEY (OrderId)
)
```

OrderId	ProductName	OrderDate
0		2011-08-06 11:13:40
1	Jarlsberg Cheese	2011-08-06 11:19:17

```
INSERT INTO OrderCheese VALUES (1, 'Jarlsberg Cheese', NOW())
```

```
SELECT ProductName, UnitPrice, NOW() as PerDate FROM Products
```

ProductName	UnitPrice	PerDate
Jarlsberg	10	2011-08-09 12:26:59
Mascarpone	33	2011-08-09 12:26:59
Gorgonzola	16	2011-08-09 12:26:59

MySQL CURDATE() Function

- CURDATE() returns the current date

```
SELECT NOW(), CURDATE(), CURTIME()
```

NOW()	CURDATE()	CURTIME()
2011-08-06 10:58:37	2011-08-06	10:58:37

```
CREATE TABLE OrderCheese
(
OrderId int NOT NULL,
ProductName varchar(50) NOT NULL,
OrderDate datetime NOT NULL,
PRIMARY KEY (OrderId)
)
```

OrderId	ProductName	OrderDate
0		2011-08-06 11:13:40
1	Jarlsberg Cheese	2011-08-06 11:19:17
2	Jarlsberg Cheese	2011-08-06 00:00:00

```
INSERT INTO OrderCheese VALUES (2, 'Jarlsberg Cheese',
CURDATE())
```

MYSQL DATE() FUNCTION

- DATE() function extracts the date part of a date or date/time expression

```
DATE(date)
```

```
SELECT ProductName, DATE(OrderDate) as OrderDate
FROM OrderCheese
WHERE OrderId=1
```

ProductName	OrderDate
Jarlsberg Cheese	2011-08-06

MySQL EXTRACT() Function

- The EXTRACT() function returns a single part of a date/time, such as year, month, day, hour, minute, etc.
- Unit value can be: SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR, or any combination of them

```
EXTRACT(unit FROM date)
```

```
SELECT EXTRACT(YEAR FROM OrderDate) as OrderYear,
EXTRACT(MONTH FROM OrderDate) as OrderMonth,
EXTRACT(DAY FROM OrderDate) as OrderDay
FROM OrderCheese
WHERE OrderID=1
```

OrderYear	OrderMonth	OrderDay
2011	8	6

MYSQL DATE_ADD() FUNCTION

- The DATE_ADD() function adds a specified time interval to a date.

```
DATE_ADD(date, INTERVAL expr type)
```

```
SELECT OrderID, DATE_ADD(OrderDate, INTERVAL 45 DAY) AS OrderPayDate
FROM OrderCheese
```

OrderID	OrderPayDate
0	2011-09-20 11:13:40
1	2011-09-20 11:19:17
2	2011-09-20 00:00:00

MYSQL DATE_SUB() FUNCTION

- The DATE_SUB() function subtracts a specified time interval from a date.

```
DATE_SUB(date, INTERVAL expr type)
```

```
SELECT OrderID, DATE_SUB(OrderDate, INTERVAL 5 DAY) AS SubtractDate
FROM OrderCheese
```

OrderID	SubtractDate
0	2011-08-01 11:13:40
1	2011-08-01 11:19:17
2	2011-08-01 00:00:00

MYSQL DATEIFF() FUNCTION

- The DATEDIFF() function returns the time between two dates.

```
DATEDIFF(date1, date2)
```

```
SELECT DATEDIFF ( '2010-11-30', '2010-11-29') AS DiffDate
```

DiffDate
1

MYSQL DATE_FORMAT() FUNCTION

- The DATE_FORMAT() function displays the date/time in different formats.

```
DATE_FORMAT(date, format)
```

```
SELECT DATE_FORMAT(NOW(), ‘%m-%d-%y’)
```

DATE_FORMAT(NOW(), '%m-%d-%y')
08-08-11

SQL DATE FUNCTIONS

- Introduces some build-in functions to deal with dates

SQL Server Date Functions

Function	Description
GETDATE()	Returns the current date and time
DATEPART()	Returns a single part of a date/time
DATEADD()	Adds or subtracts a specified time interval from a date
DATEDIFF()	Returns the time between two dates
CONVERT()	Displays date/time data in different formats

SQL Date Datatypes

- MySQL
  - Date: YYYY-MM-DD
  - Datetime: YYYY-MM-DD HH:MM:SS
  - Timestamp: YYYY-MM-DD HH:MM:SS
  - Year: YYYY or YY

```
SELECT * FROM OrderCheese WHERE OrderDate='2011-08-06'
```

OrderId	ProductName	OrderDate
2	Jarlsberg Cheese	2011-08-06 00:00:00

SQL NULL Values

- NULL values represent missing unknown data.
- NULL is used as a placeholder for unknown or inapplicable values
- NULL vs. 0

```
INSERT INTO persons VALUES (5, 'Ding', 'Ying', NULL, NULL);
```

```
SELECT * FROM Persons
WHERE Address IS NULL
```

P_Id	LastName	FirstName	Address	City
5	Ding	Ying	(NULL)	(NULL)

```
SELECT * FROM Persons
WHERE Address IS NOT NULL
```

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn10	Sandnes
2	Svendson	Tove	Borgvn23	Sandnes
3	Pettersen	Kari	Storgt20	Stavanger
4	Nilsen	Tom	Vingvn23	Stavanger

```
DELETE FROM Persons WHERE P_ID=5
```

SQL NULL FUNCTIONS

```
SELECT ProductName, UnitPrice*(UnitsInStock+UnitsOnOrder)
FROM Products
```

ProductName	UnitPrice*(UnitsInStock+UnitsOnOrder)
Jarlsberg	310
Mascarpone	(NULL)
Gorgonzola	464

ISNULL(), NVL(), IFNULL(), COALESCE() are used to treat NULL values.
Below all makes nulls to be zero

```
SELECT ProductName, UnitPrice*(UnitsInStock+ISNULL(UnitsOnOrder,0))
FROM Products
```

SQL Server/Access

```
SELECT ProductName, UnitPrice*(UnitsInStock+NVL(UnitsOnOrder,0))
FROM Products
```

Oracle

```
SELECT ProductName, UnitPrice*(UnitsInStock+IFNULL(UnitsOnOrder,0)) AS Price
FROM Products
```

MySQL

```
SELECT ProductName, UnitPrice*(UnitsInStock+COALESCE(UnitsOnOrder,0))AS Price
FROM Products
```

ProductName	Price
Jarlsberg	310
Mascarpone	759
Gorgonzola	464

SQL DATATYPE

Access

Data type	Description	Storage
Text	Use for text or combinations of text and numbers. 255 characters maximum	
Memo	Memo is used for larger amounts of text. Stores up to 65,536 characters. **Note:** You cannot sort a memo field. However, they are searchable	
Byte	Allows whole numbers from 0 to 255	1 byte
Integer	Allows whole numbers between -32,768 and 32,767	2 bytes
Long	Allows whole numbers between -2,147,483,648 and 2,147,483,647	4 bytes
Single	Single precision floating-point. Will handle most decimals	4 bytes
Double	Double precision floating-point. Will handle most decimals	8 bytes
Currency	Use for currency. Holds up to 15 digits of whole dollars, plus 4 decimal places. **Tip:** You can choose which country's currency to use	8 bytes
AutoNumber	AutoNumber fields automatically give each record its own number, usually starting at 1	4 bytes
Date/Time	Use for dates and times	8 bytes
Yes/No	A logical field can be displayed as Yes/No, True/False, or On/Off. In code, use the constants True and False (equivalent to -1 and 0). **Note:** Null values are not allowed in Yes/No fields	1 bit
OLE Object	Can store pictures, audio, video, or other BLOBs (Binary Large OBjects)	up to 1GB
Hyperlink	Contain links to other files, including web pages	
Lookup Wizard	Let you type a list of options, which can then be chosen from a drop-down list	4 bytes

SQL DATATYPE

MySQL: Text

Data type	Description
CHAR(size)	Holds a fixed length string (can contain letters, numbers, and special characters). The fixed size is specified in parenthesis. Can store up to 255 characters
VARCHAR(size)	Holds a variable length string (can contain letters, numbers, and special characters). The maximum size is specified in parenthesis. Can store up to 255 characters. **Note:** If you put a greater value than 255 it will be converted to a TEXT type
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT	Holds a string with a maximum length of 65,535 characters
BLOB	For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters
LONGBLOB	For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data
ENUM(x,y,z,etc.)	Let you enter a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted.**Note:** The values are sorted in the order you enter them. You enter the possible values in this format: ENUM('X','Y','Z')
SET	Similar to ENUM except that SET may contain up to 64 list items and can store more than one choice

SQL DATATYPE

MySQL: Number

Data type	Description
TINYINT(size)	-128 to 127 normal. 0 to 255 UNSIGNED*. The maximum number of digits may be specified in parenthesis
SMALLINT(size)	-32768 to 32767 normal. 0 to 65535 UNSIGNED*. The maximum number of digits may be specified in parenthesis
MEDIUMINT(size)	-8388608 to 8388607 normal. 0 to 16777215 UNSIGNED*. The maximum number of digits may be specified in parenthesis
INT(size)	-2147483648 to 2147483647 normal. 0 to 4294967295 UNSIGNED*. The maximum number of digits may be specified in parenthesis
BIGINT(size)	-9223372036854775808 to 9223372036854775807 normal. 0 to 18446744073709551615 UNSIGNED*. The maximum number of digits may be specified in parenthesis
FLOAT(size,d)	A small number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter
DOUBLE(size,d)	A large number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter
DECIMAL(size,d)	A DOUBLE stored as a string , allowing for a fixed decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter

Unsigned: integer cannot be negative

SQL DATATYPE

Data type	Description
DATE()	A date. Format: YYYY-MM-DD**Note:** The supported range is from '1000-01-01' to '9999-12-31'
DATETIME()	*A date and time combination. Format: YYYY-MM-DD HH:MM:SS**Note:** The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'
TIMESTAMP()	*A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD HH:MM:SS**Note:** The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC
TIME()	A time. Format: HH:MM:SS**Note:** The supported range is from '-838:59:59' to '838:59:59'
YEAR()	A year in two-digit or four-digit format.**Note:** Values allowed in four-digit format: 1901 to 2155. Values allowed in two-digit format: 70 to 69, representing years from 1970 to 2069

DATABASE

- Table :
  - student(Sno, Sname, Sgender, Sage, Sdept)
    - ✓Key
  - course(Cno, Cname, Cpno, Ccredit)
    - ✓Cpno: prerequisite course
  - enrollment(Sno, Cno, Grade)
    - ✓Students enroll courses

CREATE TABLE STUDENT

Sno	Sname	Sgender	Sage	Sdept
001	John	M	20	CS
002	Ahn	F	19	CS
003	Lily	F	18	MA
004	Sam	M	19	IS

```
CREATE TABLE student (
        Sno             INT,
        Sname           Varchar(10),
        Sgender         Varchar (1),
        Sage            INT,
        Sdept           Varchar (2),
        PRIMARY KEY (Sno)   );
INSERT INTO student VALUES (001, 'John', 'M', 20, 'CS');
INSERT INTO student VALUES (002, 'Ahn', 'F', 19, 'CS');
INSERT INTO student VALUES (003, 'Lily', 'F', 18, 'MA');
INSERT INTO student VALUES (004, 'Sam', 'M', 19, 'IS');
```

DATABASE-TABLE "COURSE"

Cno	Cname	Cpno	Ccredit
1	Databases	5	4
2	Mathematics		2
3	Information System	1	4
4	Operation System	6	3
5	Data Structure	7	4
6	Data Processing		2
7	C++	6	4

CREATE TABLE COURSE

```
CREATE TABLE course (
        Cno             INT,
        Cname           Varchar(20),
        Cpno            INT,
        Ccredit         INT,
        PRIMARY KEY (Cno)       );
INSERT INTO course VALUES (1, 'Databases', 5, 4);
INSERT INTO course VALUES (2, 'Mathematics', NULL, 2);
INSERT INTO course VALUES (3, 'Information System', 1, 4);
INSERT INTO course VALUES (4, 'Operation System', 6, 3);
INSERT INTO course VALUES (5, 'Data Structure', 7, 4);
INSERT INTO course VALUES (6, 'Data Processing', NULL, 2);
INSERT INTO course VALUES (7, 'C++', 6, 4);
```

DATABASE-TABLE ENROLLMENT

Sno	Cno	Grade
001	1	92
001	2	85
001	3	88
002	2	90
002	3	80

CREATE TABLE ENROLLMENT

```
CREATE TABLE enrollment (
    Sno      INT,
    Cno      INT,
    Grade    INT,
    PRIMARY KEY (Sno, Cno),
    FOREIGN KEY (Sno) REFERENCES student(Sno),
    FOREIGN KEY (Cno) REFERENCES course(Cno)
);

INSERT INTO enrollment VALUES (001, 1, 92);
INSERT INTO enrollment VALUES (001, 2, 85);
INSERT INTO enrollment VALUES (001, 3, 88);
INSERT INTO enrollment VALUES (002, 2, 90);
INSERT INTO enrollment VALUES (002, 3, 80);
```

JOINT QUERY

Simple Query

Self join

Outer join

Complex Query

SIMPLE QUERY

[Eg1] Query all of the students and their course enrollment

```
SELECT  student.*,  enrollment.*
FROM    student,  enrollment
WHERE  student.Sno = enrollment.Sno;
```

Sno	Sname	Sgender	Sage	Sdept	Sno	Cno	Grade
1	John	M	20	CS	1	1	92
1	John	M	20	CS	1	2	85
1	John	M	20	CS	1	3	88
2	Ahn	F	19	CS	2	2	90
2	Ahn	F	19	CS	2	3	80

SIMPLE QUERY

[Eg2] Another way to query Eg1.

```
SELECT  student.Sno,  Sname,  Sgender,  Sage,  Sdept,  Cno,  Grade
 FROM   student,  enrollment
 WHERE  student.Sno = enrollment.Sno;
```

Sno	Sname	Sgender	Sage	Sdept	Cno	Grade
1	John	M	20	CS	1	92
1	John	M	20	CS	2	85
1	John	M	20	CS	3	88
2	Ahn	F	19	CS	2	90
2	Ahn	F	19	CS	3	80

JOINT QUERY

Simple Query

Self join

Outer join

Complex Query

SELF JOIN

- Self join: Join with a table itself
- Need the alias to differentiate the table.

[Eg3] Query all of the prerequisite of each course.

```
SELECT  FIRST.Cno,  SECOND.Cpno
FROM  course  FIRST,  course  SECOND
WHERE FIRST.Cpno = SECOND.Cno;
```

+ Options

Cno	Cpno
1	7
3	5
4	*NULL*
5	6
7	*NULL*

JOINT QUERY

Simple Query
Self join
Outer join
Complex Query

OUTER JOIN

[Eg 4] Left outer join.

```
SELECT student.Sno, Sname, Sgender, Sage, Sdept, Cno, Grade
FROM student LEFT OUTER JOIN enrollment ON student.Sno=enrollment.Sno;
```

Sno	Sname	Sgender	Sage	Sdept	Cno	Grade
1	John	M	20	CS	1	92
1	John	M	20	CS	2	85
1	John	M	20	CS	3	88
2	Ahn	F	19	CS	2	90
2	Ahn	F	19	CS	3	80
3	Lily	F	18	MA	*NULL*	*NULL*
4	Sam	M	19	IS	*NULL*	*NULL*

OUTER JOIN

[Eg 41] Right outer join.

```
SELECT student.Sno, Sname, Sgender, Sage, Sdept, Cno, Grade
FROM student RIGHT OUTER JOIN enrollment ON student.Sno=enrollment.Sno;
```

Sno	Sname	Sgender	Sage	Sdept	Cno	Grade
1	John	M	20	CS	1	92
1	John	M	20	CS	2	85
1	John	M	20	CS	3	88
2	Ahn	F	19	CS	2	90
2	Ahn	F	19	CS	3	80

JOINT QUERY

Simple Query
Self join
Outer join
Complex Query

COMPLEX QUERY

[Eg5] Query all of the students who have enrolled course 2 with a grade 90 or more

```
SELECT student.Sno,  Sname
FROM    student,  enrollment
WHERE student.Sno = enrollment.Sno AND
       enrollment.Cno=2 AND enrollment.Grade >= 90;
```

Sno	Sname
2	Ahn

COMPLEX QUERY

[Eg6] Query all of the students' id, name, enrolled courses, and their grades.

```
SELECT student.Sno, Sname, Cname, Grade
FROM    student, enrollment, course
WHERE student.Sno = enrollment.Sno
     and enrollment.Cno = course.Cno;
```

Sno	Sname	Cname	Grade
1	John	Databases	92
1	John	Mathematics	85
1	John	Information System	88
2	Ahn	Mathematics	90
2	Ahn	Information System	80

NESTED QUERY

- Nested query
  - Query Component: A SELECT-FROM-WHERE statement
  - Nested query: A query component is in another WHERE or HAVING clause

NESTED QUERY

```
SELECT Sname
FROM student
WHERE Sno IN
        (SELECT Sno
            FROM enrollment
            WHERE Cno=2);
```

NESTED QUERY – IN

[Eg7] Query the students who are in the same department with Ahn.

```
SELECT Sno, Sname, Sdept
FROM student
WHERE Sdept  IN
      (SELECT Sdept
        FROM student
        WHERE Sname= 'Ahn');
```

Sno	Sname	Sdept
1	John	CS
2	Ahn	CS

NESTED QUERY – IN

[Eg8] Using self-join to query Eg7.

```
SELECT  S1.Sno, S1.Sname, S1.Sdept
FROM     student S1, student S2
WHERE  S1.Sdept = S2.Sdept  AND S2.Sname = 'Ahn';
```

Sno	Sname	Sdept
1	John	CS
2	Ahn	CS

NESTED QUERY – IN

[Eg9] Query all of the id and the names of the students who have enrolled the course “Information System”.

```
SELECT Sno, Sname
            FROM    student
            WHERE Sno  IN
          (SELECT Sno
           FROM    enrollment
           WHERE  Cno IN
                (SELECT Cno
                 FROM course
                 WHERE Cname= ‘Information System’
                )
          );
```

Sno	Sname
1	John
2	Ahn

NESTED QUERY – IN

[Eg10] Query Eg9 using joint query.

```
SELECT student.Sno, Sname
FROM    student, enrollment, course
WHERE student.Sno = enrollment.Sno  AND
      enrollment.Cno = course.Cno AND
      course.Cname='Information System';
```

Sno	Sname
1	John
2	Ahn

NESTED QUERY – COMPARISON OPERATOR

[Eg11] Suppose that a student can be only in ONE department, we can use "=" in Eg10.

```
SELECT Sno, Sname, Sdept
    FROM    student
    WHERE Sdept   =
                (SELECT Sdept
                 FROM    student
                 WHERE Sname='Ahn');
```

Sno	Sname	Sdept
1	John	CS
2	Ahn	CS

NESTED QUERY – COMPARISON OPERATOR

［Eg12］Query all of the course id in which the student was graded with a score more than his/her average grade.

```
SELECT Sno,  Cno
   FROM  enrollment  x
   WHERE Grade >=(SELECT AVG(Grade)
                    FROM  enrollment y
                   WHERE y.Sno=x.Sno);
```

Sno	Cno
1	1
2	2

NESTED QUERY – COMPARISON OPERATOR

［Eg121］Query all of the course id in which the student was graded with a score more than his/her average grade and show student name

```
SELECT x.Sno,  Cno, Sname
   FROM  enrollment  x, student
   WHERE x.Sno=student.Sno and
            Grade >=(SELECT AVG(Grade)
                        FROM  enrollment y
                       WHERE y.Sno=x.Sno);
```

Sno	Cno	Sname
1	1	John
2	2	Ahn

NESTED QUERY – ANY AND ALL

> ANY
> ALL
< ANY
< ALL
>= ANY
>= ALL
<= ANY
<= ALL
= ANY
=ALL (seldom used)
!=（or <>）ANY
!=（or <>）ALL

NESTED QUERY – ANY AND ALL

[Eg13] Query all of the names and ages of the students who are younger than any student in CS department. These students should not in the CS department.

```
SELECT Sname, Sage
   FROM    student
   WHERE Sage < ANY (SELECT  Sage
                          FROM    student
                          WHERE Sdept= 'CS')
        AND Sdept <> 'CS';
```

Sname	Sage
Lily	18
Sam	19

NESTED QUERY – ANY AND ALL

[Eg14] Query Eg13 using aggregate function.

```
SELECT Sname, Sage
   FROM   student
   WHERE Sage <
                    (SELECT MAX(Sage)
                     FROM student
                     WHERE Sdept= 'CS')
       AND Sdept <> 'CS';
```

Sname	Sage
Lily	18
Sam	19

NESTED QUERY – ANY AND ALL

[Eg15] Query all of the names and ages of the students who are younger than all of the students in CS department. Note that these students should be not in CS department.

Solution 1: Using ALL

```
SELECT Sname, Sage
FROM student
WHERE Sage < ALL
        (SELECT Sage
         FROM student
         WHERE Sdept= 'CS')
       AND Sdept <> 'CS';
```

Sname	Sage
Lily	18

NESTED QUERY – ANY AND ALL

Solution 2: Using aggregate function

```
SELECT Sname, Sage
FROM student
WHERE Sage <
          (SELECT MIN(Sage)
           FROM student
           WHERE Sdept='CS')
     AND Sdept <> 'CS';
```

Sname	Sage
Lily	18

NESTED QUERY – ANY AND ALL

ANY, ALL, IN, and aggregate function

	=	<> or !=	<	<=	>	>=
ANY	IN	--	<MAX	<=MAX	>MIN	>= MIN
ALL	--	NOT IN	<MIN	<= MIN	>MAX	>= MAX

NESTED QUERY – EXISTS

[Eg16] Query all of the names of the students who have enrolled course 1.

```
SELECT Sname
    FROM student
    WHERE EXISTS
            (SELECT *
             FROM enrollment
             WHERE Sno=student.Sno AND Cno= 1);
```

Sname
John

NESTED QUERY – EXISTS

[Eg17] Query Eg16 using joint query.

```
SELECT Sname
FROM student,  enrollment
WHERE student.Sno=enrollment.Sno AND enrollment.Cno= 1;
```

Sname
John

NESTED QUERY – EXISTS

[Eg18] Query all of the names of the students who have never enrolled course 1.

```
SELECT Sname
    FROM student
    WHERE NOT EXISTS
            (SELECT *
             FROM enrollment
             WHERE Sno = student.Sno AND Cno=1);
```

Sname
Ahn
Lily
Sam

NESTED QUERY – EXISTS

[Eg19] Query all of the students who are in the same department as Ahn.

```
SELECT Sno, Sname, Sdept
    FROM student S1
     WHERE EXISTS
        (SELECT *
         FROM student S2
         WHERE S2.Sdept = S1.Sdept AND
             S2.Sname ='Ahn');
```

Sno	Sname	Sdept
1	John	CS
2	Ahn	CS

NESTED QUERY – EXISTS

- When using EXISTS/NOT EXISTS
  - There is no IMPLICATION in SQL
  - However, we can transform as:

$$p \rightarrow q \equiv \neg\, pVq$$

NESTED QUERY – EXISTS

[Eg20] Query all of the id of the students who have enrolled at least all of the courses student 002 had enrolled.

- Query the student with an “x” number id
- For all of the course “y”, if student 002 enrolled y, then x should enroll y.
- Annotation:
  - p: student 002 enrolled course y.
  - q: student x enrolled course y.
  - $(\forall y)\ p \rightarrow q$

Nested Query – Exists

- Transformation:

$$(\forall y)p \rightarrow q \equiv \neg (\exists y (\neg(p \rightarrow q)))$$
$$\equiv \neg (\exists y (\neg(\neg p \vee q)))$$
$$\equiv \neg \exists y(p \wedge \neg q)$$

- Meaning: There is no such course y that student 002 enrolled but student x did not.

NESTED QUERY – EXISTS

```
SELECT DISTINCT Sno
      FROM enrollment SCX
      WHERE NOT EXISTS
                (SELECT *
                 FROM enrollment SCY
                 WHERE SCY.Sno = 002  AND
                           NOT EXISTS
                           (SELECT *
                            FROM enrollment SCZ
                            WHERE SCZ.Sno=SCX.Sno AND
                                      SCZ.Cno=SCY.Cno));
```

Sno
1
2

SUMMARY: SQL STATEMENT

SELECT [ALL|DISTINCT]

<column> [alias] [, <column> [alias]] …

FROM <table> [alias]

[, < table > [alias]] …

[WHERE < conditional expression >]

[GROUP BY <column>

[HAVING < conditional expression >]]

[ORDER BY <column2> [ASC|DESC]

2) JSP & SERVLET

SERVLETS

- The purpose of a servlet is to create a Web page in response to a client request
- Servlets are written in Java, with a little HTML mixed in
  - The HTML is enclosed in out.println( ) statements
- JSP (Java Server Pages) is an alternate way of creating servlets
  - JSP is written as ordinary HTML, with a little Java mixed in
  - The Java is enclosed in special tags, such as <% … %>
  - The HTML is known as the template text
- JSP files must have the extension .jsp
  - JSP is *translated* into a Java servlet, which is then *compiled*
  - Servlets are run in the usual way
  - The browser or other client sees only the resultant HTML, as usual
- Tomcat knows how to handle servlets and JSP pages

JAVA SERVER PAGES (JSP)

- Java Server Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications. JSP have access to the entire family of Java APIs, including the JDBC API to access enterprise databases.

- JSP is dynamic web page
- –JSP is written as ordinary HTML, with a little Java mixed
- –The Java is enclosed in special tags, such as <%...%>
- –JSP files must have the extension .jsp
- JSP is *translated* in to a Javaservlet ,which is then *compiled*

How JSP Works

- When Tomcat needs to use a JSP page, it:
  - Translates the JSP into a Java servlet
  - Compiles the servlet
  - Creates one instance of the JSP servlet
  - Executes the servlet as normal Java code
  - Hence, when you are writing JSP, you are writing “higher-level” Java code
- Each call to the JSP servlet is executed in a new Thread
  - Since there is only one JSP object, you have to use synchronization if you use any instance variables of the servlet
- You have two basic choices when using JSP:
  - Let the JSP do all the work of a servlet
  - Write a servlet that does the work and passes the results to JSP to create the resultant HTML page
    - This works because a servlet can call another servlet
- Bottom line: JSP is just a convenient way of writing Java code!

ADVANTAGES OF JSP

- Build process is performed automatically .
- Translation phase treat each data type in java Server Pages differently.
- Template data is transformed into code .
  - This code emits that data stream that returns data to the client.
  - It is faster than Servlet.
- Can embed java coding in it.

THE PROCESSING OF JSP

- When the browser asks the Web server for a JSP, the Web server passes control to a JSP container .

- A container works with the Web server to provide the runtime environment and other services a JSP needs .

- It knows how to understand the special elements that are part of JSPs . Because this is the first time this JSP has been invoked , the JSP container converts it into an executable unit called a Servlet .

CONT'D

- The entire page, including the parts that are in HTML, is translated into source code . After code translation, the JSP container compiles the Servlet , loads it automatically, and then executes it.

- Typically, the JSP container checks to see whether a Servlet for a JSP file already exists .
  - If not it does the translation process
  - If version does not match do the translation process

JSP LIFE CYCLE

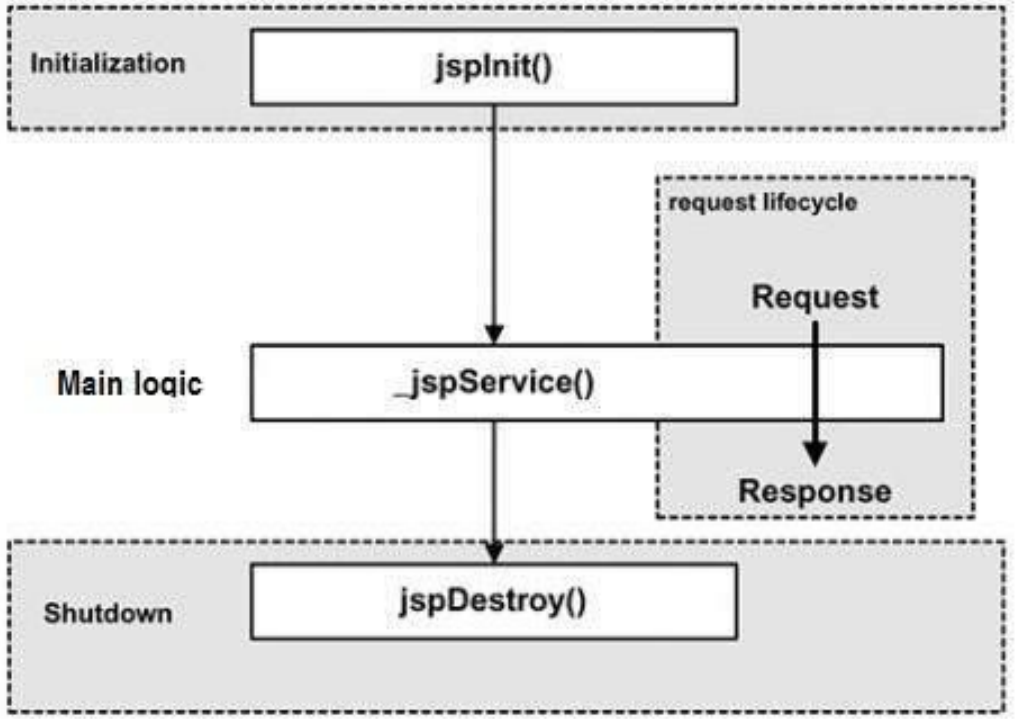
- Some how similar to Servlet life cycle. When any request mapped to a JSP, it is handled by a
- special Servlet event.
- Servlet checks the java Server Page's Servlet is older than the java Server Page .

- If so it translates to java Server Page into a Servlet class & compiles the class.
- Next the Servlet sends response to the java server page.

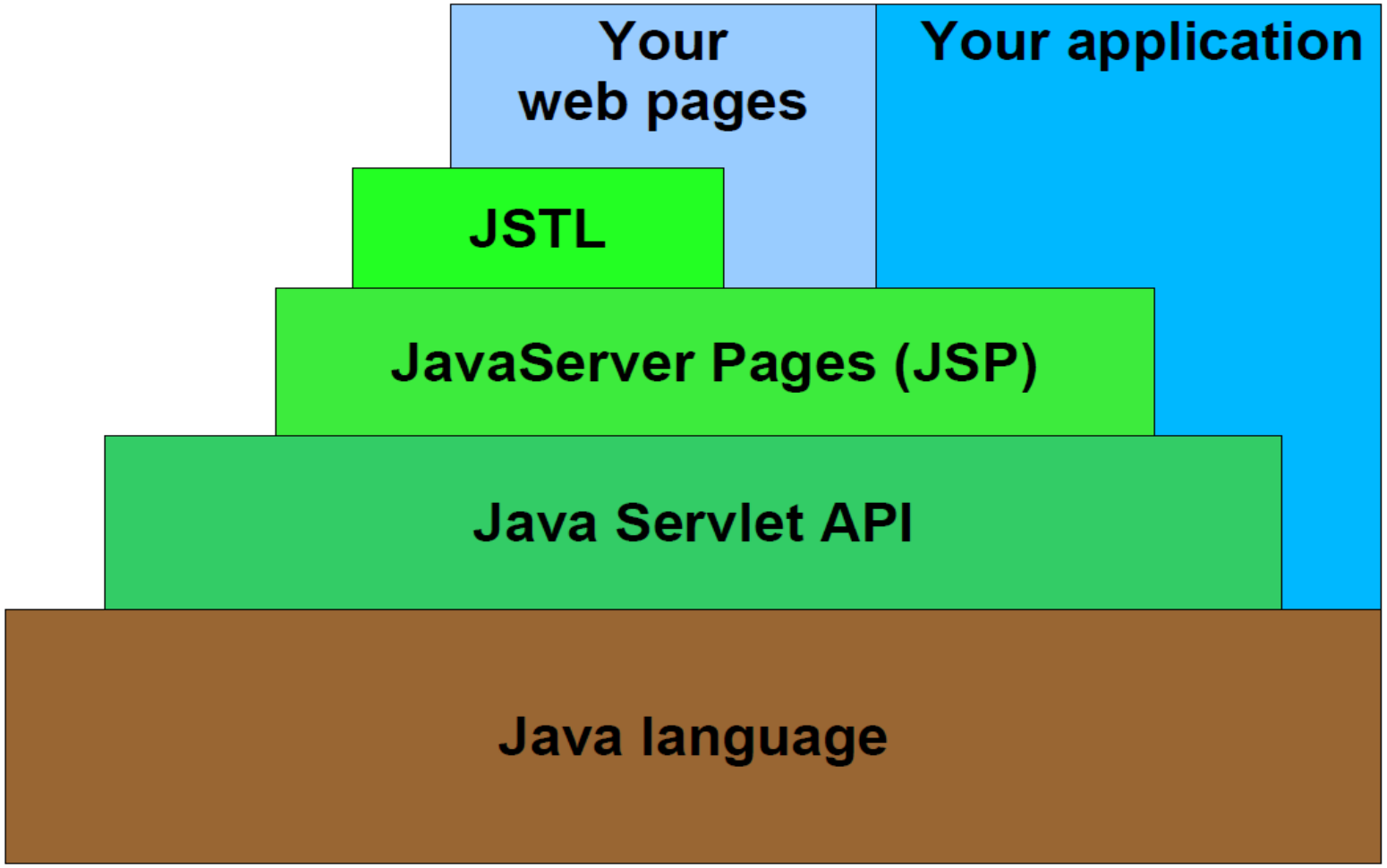
JSP LIFE CYCLE

The paths followed by a JSP in it's lifecycle:

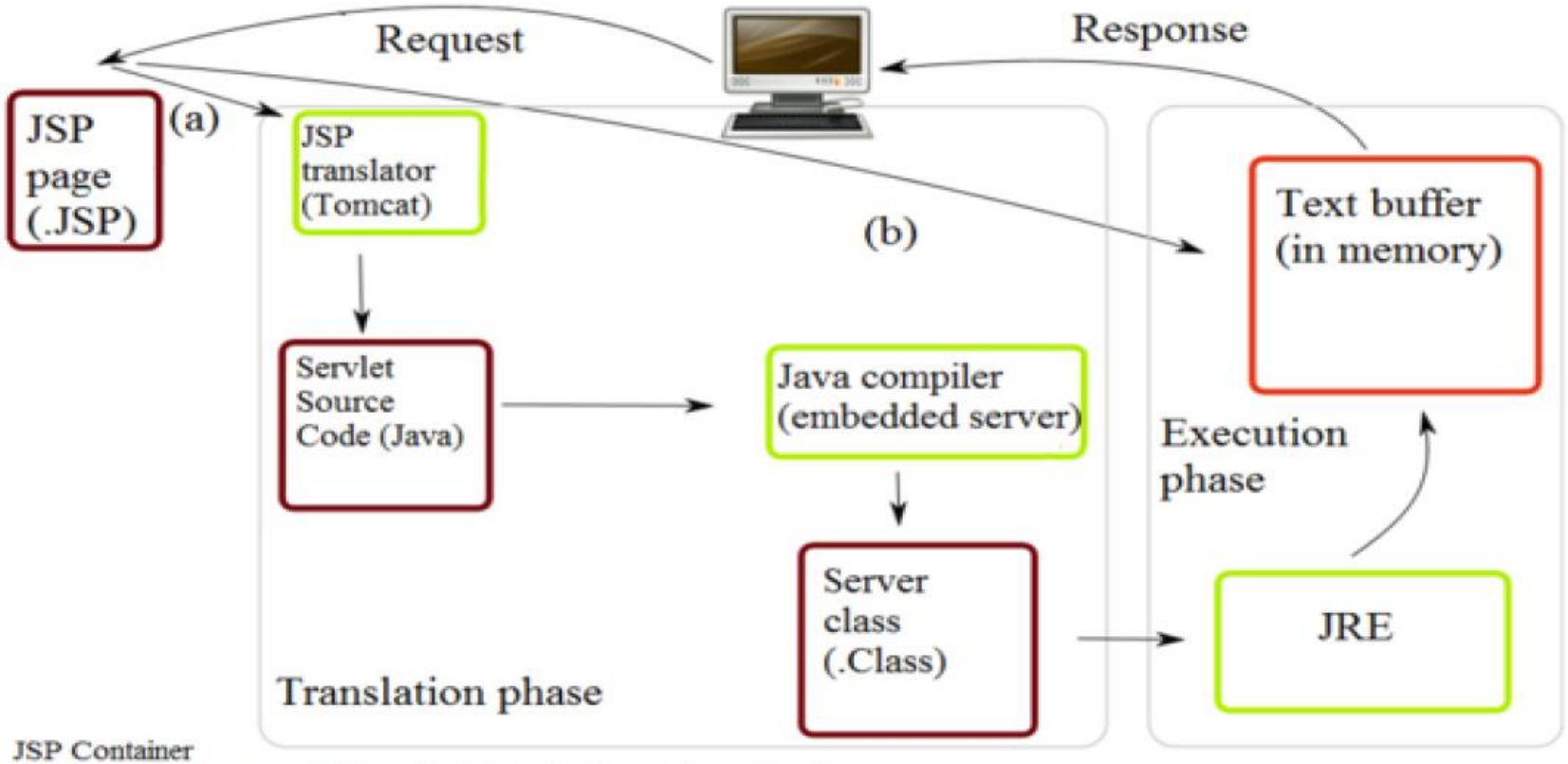
- Compilation
- Initialization
- Execution
- Cleanup
- The four major phases of a JSP life cycle are very similar to the Servlet Life Cycle.



ORGANIZATION OF THE PLATFORM



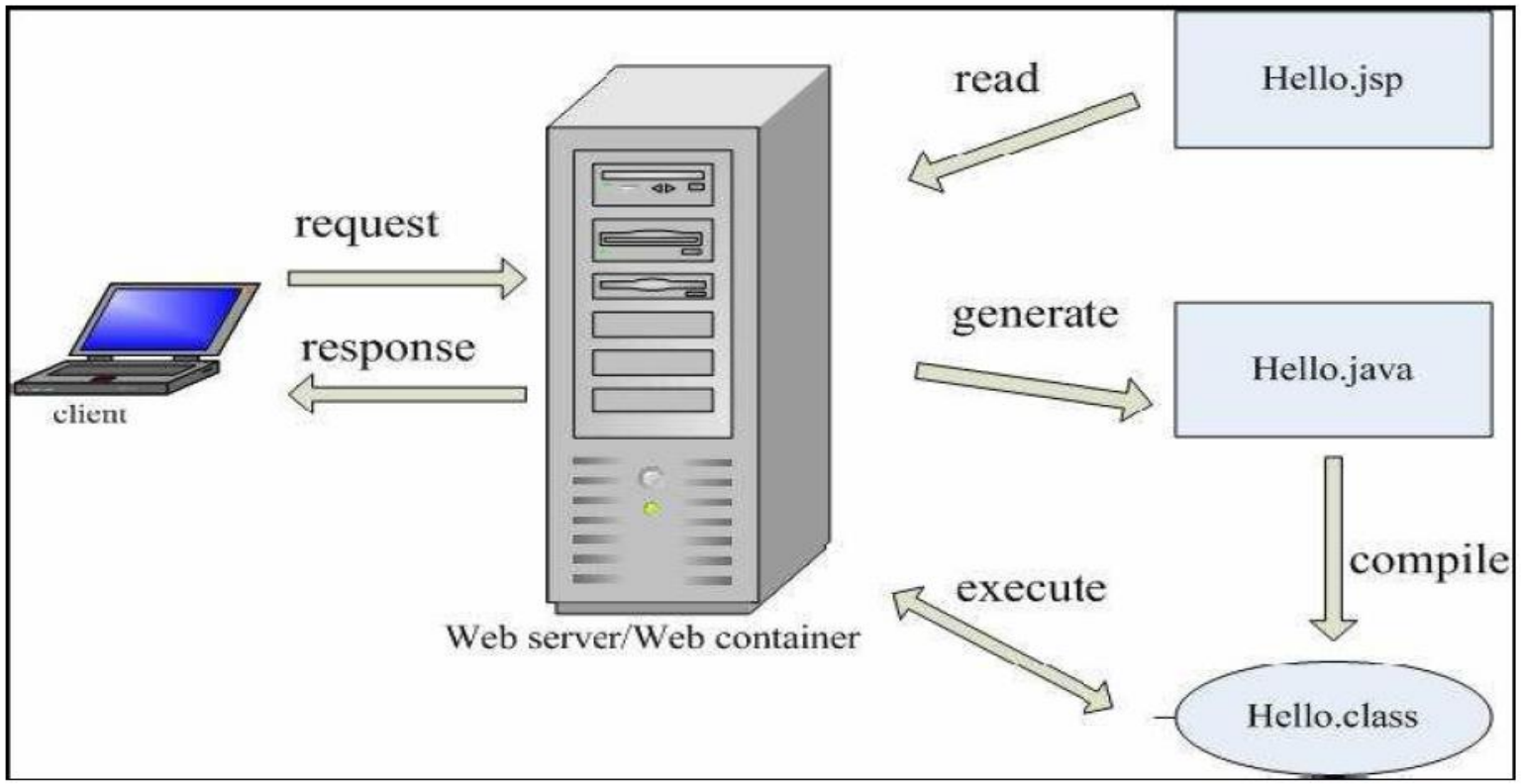
JSP ENVIRONMENT



JSP Container
(a) Translation occurs at this point, if JSP has been changed or is new.
(b) If not, translation is skipped.

JSP FLOW

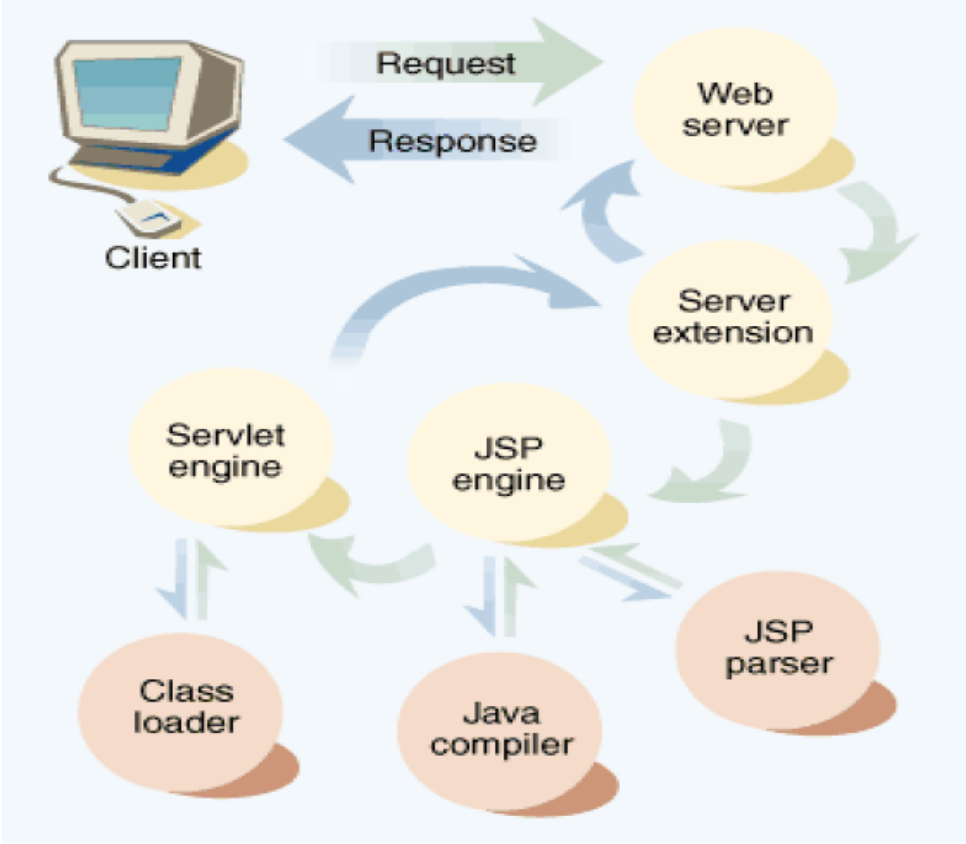
- JSP pages “live” within a container that manages its interaction:
  - HTTP Protocol (request, response, header)
  - Sessions



LOOK AT THE FLOW

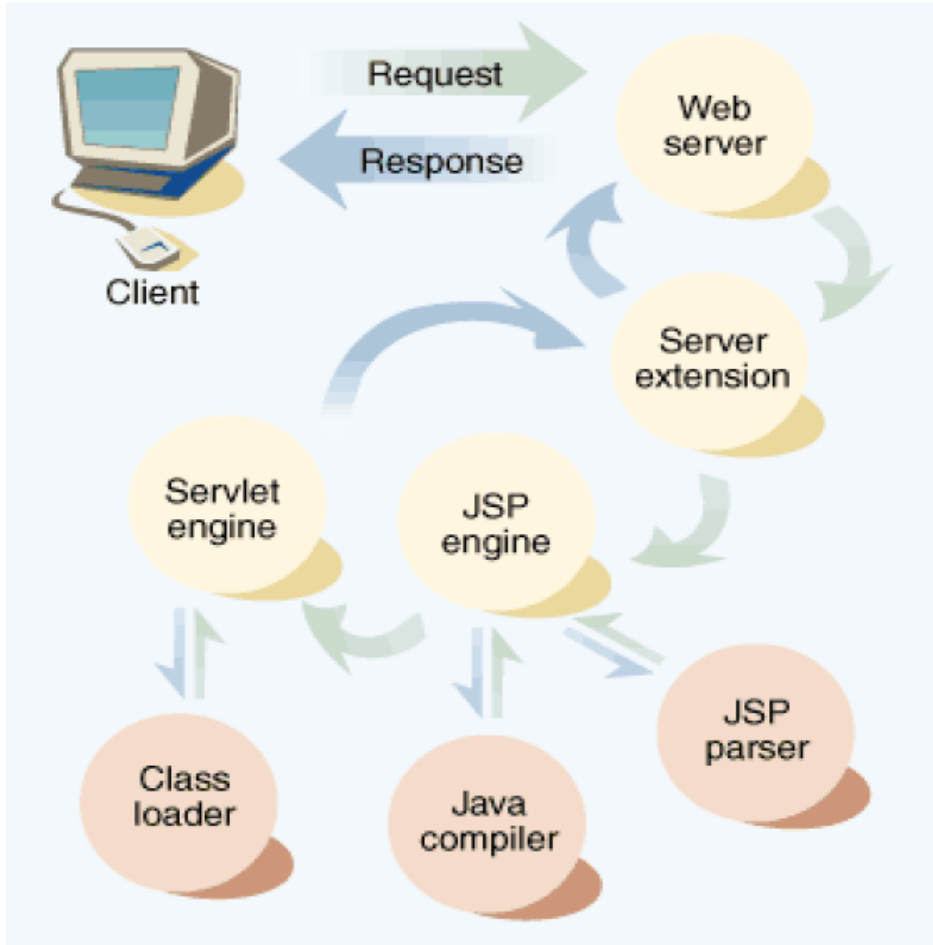
How it really works... (1/2)

- Client requests a page ending with ".jsp"
- Web Server fires up the JSP engine
- JSP engine checks whether JSP file is new or changed
- JSP engine converts the page into a Java servlet (JSP parser)
- JSP engine compiles the servlet (Java compiler)



How it really works... (2/2)

- Servlet Engine executes the new Java servlet using the standard API
- Servlet's output is transferred by Web Server as a http response



JSP TAGS

A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language.

- •<%=expression%>
- –The ***expression*** is evaluated and the result is inserted into the HTML page
- •<%code%>
- –The ***code*** is inserted into the servlet's service method
  - If ***code*** contains declarations, they become *local* variables of the service method
- –This construction is called a scriptlet
- •<%!declarations%>
- –The ***declarations*** are inserted into the servlet *class*, not into a method

STRUCTURE OF A JSP FILE

- Similar to a HTML document. Four basic tags:
  - Scriplet
  - Expression
  - Declaration
  - Definition

```
<html>
  <head>
    <title>Hello World</head>
  </head>
<body>
<h1>Hello, World</h1>
It's <%=(new java.util.Date()).toString()%>
and all is well.
</body>
</html>
```

Example JSP

- ```
  <HTML>
  <BODY>
  Hello!  The time is now
  </BODY>
  </HTML>
  ```

- Notes:
  - The `<%= ... %>` tag is used, because we are computing a *value* and inserting it into the HTML
  - The fully qualified name (`java.util.Date`) is used, instead of the short name (`Date`), because we haven't yet talked about how to do `import` declarations
  ```

EXAMPLE JSP

- **<head>**
  <title>First JSP page.</title>
  </head>
  <body>
  ** <%=“ Web Developers Paradise"%> **
  ** <%="Hello JSP"%> **
  </body>
  </html>
  In jsp java codes are written between '<%' and '%>' tags. So it takes the following form : <%= Some Expression %> In this example we have use <%="Java Developers Paradise"%>

COMMENTS

- You can put two kinds of comments in JSP:
  - <!-- HTML comment -->
    - This is an ordinary HTML comment, and forms part of the page that you send to the user
    - Hence, the user can see it by doing View source
    - JSP scriptlets in HTML comments *will* be executed
  - <%-- JSP comment -->
    - This kind of comment will be stripped out when the JSP is compiled
    - It's intended for page developers; the user will never see it

SCRIPTLETS

- Scriptlets are enclosed in <% … %> tags
  - Scriptlets are executable code and do not directly affect the HTML
  - Scriptlets *may* write into the HTML with out.print(***value***) and out.println(***value***)
  - Example:

```
<% String queryData = request.getQueryString();
   out.println("Attached GET data: " + queryData); %>
```

- Scriptlets are inserted into the servlet *exactly as written*, and are not compiled until the entire servlet is compiled
  - Example:

```
<% if (Math.random() < 0.5) { %>
        Have a <B>nice</B> day!
<% } else { %>
        Have a <B>lousy</B> day!
<% } %>
```

The Case Against Scriptlets

- One of the principal motivations for JSP is to allow Web designers who are *not* Java programmers to get some of the features of Java into their pages
- Hence, in some cases it is desirable to put as little actual Java into your JSP as possible
- Where this is a goal, a better approach is to provide the necessary Java functionality via *methods* in a class which is loaded along with the servlet

SCRIPLET TAG

- Two forms:
  - `<% any java code %>`
  - `<jsp:scriptlet> ... </jsp:scriptlet>`
    - (XML form)
- Embeds Java code in the JSP document that will be executed each time the JSP page is processed.
- Code is inserted in the service() method of the generated Servlet

CON'T

```
<html>
<body>
   <% for (int i = 0; i < 2; i++) { %>
      <p>Hello World!</p>
   <% } %>
</body>
</html>
```

```
<html>
<body>
   <p>Hello World!</p>
   <p>Hello World!</p>
</body>
</html>
```

DICLARATION IN JSP…

- A declaration declares one or more variables or methods that you can use in Java code later in the JSP file. You must declare the variable or method before you use it in the JSP file.

- <%! int i = 0; %>
- <%! int a, b, c; %>
- <%! Circle a = new Circle(2.0); %>

Declarations...

- Use <%! … %> for declarations to be added to your servlet class, not to any particular method
  - Caution: Servlets are multithreaded, so nonlocal variables must be handled with extreme care
  - If declared with <% … %>, variables are local and OK
  - If declared with <%! … %>, variables may need to be synchronized
  - Data can also safely be put in the request or session objects
- Example:

```
<%! private int accessCount = 0; %>
      Accesses to page since server reboot:
<%= ++accessCount %>
```

- You can use <%! … %> to declare *methods* as easily as to declare *variables*

DECLARATIONS

- •Use <%! ... %>tag for declarations
- –If declared with<% ... %>, variables are local

- •Example:
- <%! int accessCount = 0; %>
- :
- <%= ++accessCount %>

- •You can use <%! ... %>to declare methods as easily as to
- declare variables

JSP EXPRESSION…

- A JSP expression element contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file.

- Because the value of an expression is converted to a String, you can use an expression within a line of text, whether or not it is tagged with HTML, in a JSP file.

- The expression element can contain any expression that is valid according to the Java Language Specification but you cannot use a semicolon to end an expression.

JSP EXPRESSION

- <jsp:expression>
- expression
- </jsp:expression>

- <html>
- <head><title>A Comment Test</title></head>
- <body>
- Today's date: <%= (new java.util.Date()).toLocaleString()%>
- </body>
- </html>

JSP COMMENTS

- •Different from HTML comments.
- •HTML comments are visible to client.

- <!--an HTML comment -->

- •JSP comments are used for documenting JSP code.
- •JSP comments are not visible client-side.

- <%--a JSP comment --%>

JSP OPERATORS

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

CONTROL FLOW IN JSP

The if...else block starts out like an ordinary Scriptlet, but the Scriptlet is closed at each line with HTML text included between the Scriptlet tags.

```
<%! int day = 3; %>
<html>
  <head><title>IF...ELSE Example</title></head>

  <body>
    <% if (day == 1 | day == 7) { %>
      <p> Today is weekend</p>
    <% } else { %>
      <p> Today is not weekend</p>
    <% } %>
  </body>
</html>
```

LOOP STATEMENT

You can also use three basic types of looping blocks in Java: for, while, and do…while blocks in your JSP programming.

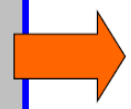
```
<%! int fontSize; %>
<html>
   <head><title>FOR LOOP Example</title></head>

   <body>
      <%for ( fontSize = 1; fontSize <= 3; fontSize++){ %>
         <font color = "green" size = "<%= fontSize %>">
            JSP Tutorial
      </font><br />
      <%}%>
   </body>
</html>
```

Expression Tag

- <%= *expr* %>
- <jsp:expression> *expr* </jsp:expression>
- Expression *expr* is evaluated (in Java) and its value is placed in the output.
  - Note: no semi-colon ";" following expr

```
<html>
<body>
<p>
<%= Integer.toString( 5 * 5 ) %>
</p>
</body>
</html>
```



```
html>
<body>
        <p>25</p>
</body>
</html>
```

(Embedded) Expression language

- An EL expression always starts with a `${` and ends with a `}`
- All EL expressions are evaluated at **runtime**
- The EL usually handles data type conversion and null values -> easy to use
- The expression can include
  - literals (“1”, “100” etc)
  - variables
  - implicit variables

Examples

- `${1+2+3}`
- `${param.Address}`

DIRECTIVES

Directives provide directions and instructions to the container, telling it how to handle certain aspects of the JSP processing.
A JSP directive affects the overall structure of the servlet class. It usually has the following form :

<%@ directive attribute = "value" %>

- Directives affect the servlet class itself
- A directive has the form:

  <%@ ***directive attribute***="***value***" %>

  or

  <%@ ***directive attribute1***="***value1***"
  attribute2="***value2***"
  ...
  attributeN="***valueN***" %>

- The most useful directive is page, which lets you import packages
  - Example: <%@ page import="java.util.*" %>

The JSP @page Directive

- `import="package.class"` or `import="pkg.class1,...,pkg.classN"`
  - This lets you specify what packages should be imported. The import attribute is the only one that is allowed to appear multiple times.
  - Example: <%@ page import="java.util.*" %>
- `contentType="MIME-Type"` or `contentType="MIME-Type; charset=Character-Set"`
  - Specifies the MIME type of the output. Default is `text/html`.
  - Example: <%@ page contentType="text/plain" %> equivalent to <% response.setContentType("text/plain"); %>

THE include DIRECTIVE

- The include directive inserts another file into the file being parsed
  - The included file is treated as just more JSP, hence it can include static HTML, scripting elements, actions, and directives
- Syntax: <%@ include file="***URL*** " %>
  - The ***URL*** is treated as relative to the JSP page
  - If the ***URL*** begins with a slash, it is treated as relative to the home directory of the Web server
- The include directive is especially useful for inserting things like navigation bars

Action Tags

- Actions are XML-syntax tags used to control the servlet engine
- <jsp:include page="***URL*** " flush="true" />
  - Inserts the indicated relative ***URL*** at *execution* time (not at compile time, like the include directive does)
  - This is great for rapidly changing data
- <jsp:forward page="***URL***" />
  <jsp:forward page="<%= ***JavaExpression*** %>" />
  - Jump to the (static) ***URL*** or the (dynamically computed) ***JavaExpression*** resulting in a URL

ACTION ELEMENTS ARE BASICALLY PREDEFINED FUNCTIONS.

1	**jsp:include** Includes a file at the time the page is requested.	6	**jsp:plugin** Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin.
2	**jsp:useBean** Finds or instantiates a JavaBean.	7	**jsp:element** Defines XML elements dynamically.
3	**jsp:setProperty** Sets the property of a JavaBean.	8	**jsp:attribute** Defines dynamically-defined XML element's attribute.
4	**jsp:getProperty** Inserts the property of a JavaBean into the output.	9	**jsp:body** Defines dynamically-defined XML element's body.
5	**jsp:forward** Forwards the requester to a new page.	10	**jsp:text** Used to write template text in JSP pages and documents.

JSP - AUTO REFRESH

Consider a webpage which is displaying live game score or stock market status or currency exchange ration. For all such type of pages, you would need to refresh your Webpage regularly using refresh or reload button with your browser.

- JSP makes this job easy by providing you a mechanism where you can make a webpage in such a way that it would refresh automatically after a given interval.
- The simplest way of refreshing a Webpage is by using the setIntHeader() method of the response object. Following is the signature of this method −

- public void setIntHeader(String header, int headerValue)

This method sends back the header "Refresh" to the browser along with an integer value which indicates time interval in seconds.

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

<html>
  <head>
    <title>Auto Refresh Header Example</title>
  </head>
    <body>
    <center>
      <h2>Auto Refresh Header Example</h2>
      <%
        // Set refresh, autoload time as 5 seconds
        response.setIntHeader("Refresh", 5);

        // Get current time
        Calendar calendar = new GregorianCalendar();
        String am_pm;

 int hour = calendar.get(Calendar.HOUR);
        int minute =
calendar.get(Calendar.MINUTE);
        int second =
calendar.get(Calendar.SECOND);

        if(calendar.get(Calendar.AM_PM) == 0)
          am_pm = "AM";
        else
          am_pm = "PM";
        String CT = hour+":"+ minute +":"+
second +" "+ am_pm;
        out.println("Crrent Time: " + CT + "\n");
      %>
    </center>

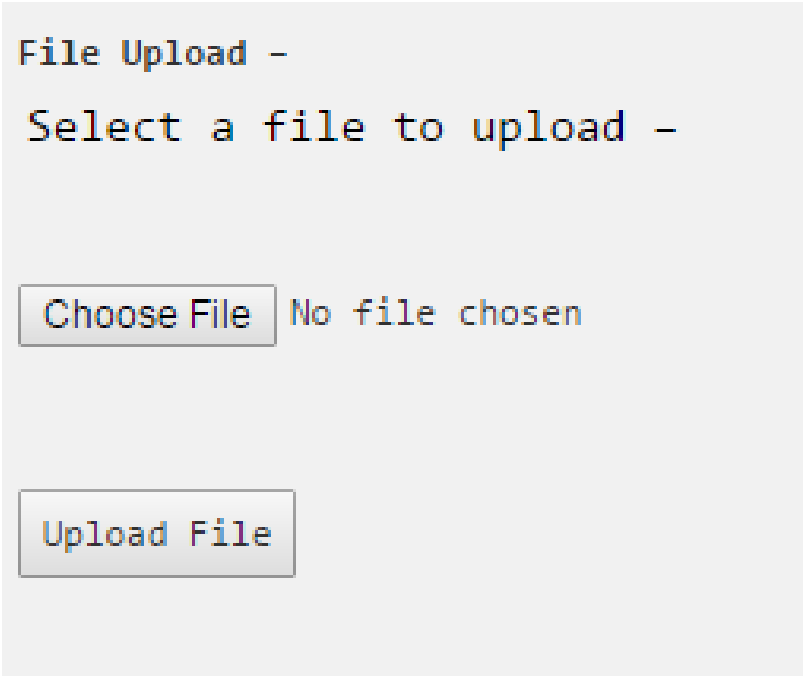
  </body>
</html>
```

FILE UPLOADING WITH JSP

```
<html>
  <head>
    <title>File Uploading Form</title>
  </head>

  <body>
    <h3>File Upload:</h3>
    Select a file to upload: <br />
    <form action = "UploadServlet" method = "post"
      enctype = "multipart/form-data">
      <input type = "file" name = "file" size = "50" />
      <br />
      <input type = "submit" value = "Upload File" />
    </form>
  </body>

</html>
```



FILE UPLOADING

- First it creates an uploader form. Following are the important steps:
- The form method attribute should be set to POST method and GET method can not be used.
- The form enctype attribute should be set to multipart/form-data.
- The form action attribute should be set to a JSP file which would handle file uploading at backend server. Following example is using uploadFile.jsp program file to upload file.
- To upload a single file you should use a single <input .../> tag with attribute type = "file". To allow multiple files uploading, include more than one input tag with different values for the name attribute. The browser associates a Browse button with each of them.

JSP IN XML

- JSP can be embedded in XML as well as in HTML
- Due to XML's syntax rules, the tags must be different (but they do the same things)
- HTML: <%= ***expression*** %>
  XML: <jsp:expression>***expression***</jsp:expression>
- HTML: <% ***code*** %>
  XML: <jsp:scriptlet>***code***</jsp:scriptlet>
- HTML: <%! ***declarations*** %>
  XML: <jsp:declaration>***declarations***</jsp:declaration>
- HTML: <%@ include file=***URL*** %>
  XML: <jsp:directive.include file="***URL***"/>

ACTIONS

- <jsp: forward page="ssParameters.jsp">
- <jsp: param name="myParam" value=“B Jena"/>
- <jsp: param name="Age" value="15"/>
- </jsp: forward>
- Name: <%= request.getParameter("myParam") %>

JSP IMPLICIT OBJECTS

- •JSP provides several implicitObject

- –request: The HttpServletRequestparameter
- –response: The HttpServletResponseparameter
- –session: The HttpSessionassociated with the request,
- or nullif there is none
- –out: A JspWriter(like a PrintWriter) used to send
- output to the client
- –application : Exist through out the application
- –exception :Show the error information

EXAMPLE (IMPLICIT OBJECT)

- Request :
- <a href="queryString.jsp?id=22&name=guru">
- request.getQueryString();
- <input type="text" name="name">
- request.getParameter("name");
- <%=request.getRequestURI()%>
- Example:
  - Your hostname: <%= request.getRemoteHost() %>
- Response :
- response.sendRedirect("http://www.google.com”);
- response.setHeader("Cache-Control","no-cache");
- response.setContentType("text/html");

SESSION IN JSP

- In session management whenever a request comes for any resource, a unique token is generated by the server and transmitted to the client by the response object and stored on the client machine as a cookie.

- **Sessionmanagement**
- **(i)SessionObject**
- **(ii)Cookies**
- **(iii)HiddenFormFields**
- **(iv)URLRewriting**

SESSION IN JSP

- Set Session Attribute
- String svalue = request.getParameter("sesvalue_txt");
- if(svalue!=null)
- {
- session.setAttribute("sesval",request.getParameter("sesvalue_txt"));
- }

SESSION IN JSP

- Using Session Attribute

- <% if(session.getAttribute("sesval")==null){ %>

- <jsp:forward page = "CreateSessionValue.jsp"/>

- <% } %>

- <h1> Hello <%= (String)session.getAttribute("sesval") %>
- </h1>

- Remove Session Attribute

- <%session.removeAttribute("sesval");%>

APPLICATION OBJECT IN JSP

- <% Integer hitsCount = (Integer)application.getAttribute("hitCounter");
- if(hitsCount ==null || hitsCount == 0){
- out.println("Welcome to my website!");
- HitsCount = 1;
- }else{
- out.println("Welcome back to my website!");
- hitsCount += 1;
- } application.setAttribute("hitCounter", hitsCount); %>
- Total number of visits: <%= hitsCount%>

Implicit Objects

- `request`
  - The HttpServletRequest parameter
  - Same usage as in servlets
  - Mainly used for **getting request parameters**
- `response`
  - The HttpServletResponse parameter
  - Same usage as in servlets
  - Rarely used in JSP (directives already to the work for us...)
- `out`
  - The **PrintWriter** associated to the **response** (buffered)
  - `out.println()`
  - Not much used... just escape to HTML
    - `%>html code<%`

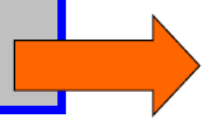
Implicit Objects

- `session`
  - The HttpSession object associated to the request
  - Same usage as in servlets
  - Created automatically
- `application`
  - The ServletContext object
  - Used to share variables across *all servlets* in the application
  - getAttribute and setAttribute methods
- `config`
  - The ServletConfig object
  - Same usage as in servlets
- `pageContext`
  - The PageContext object

Request Parameters

- JSP provides access to the *implicit* object **request** that stores attributes related to the request for the JSP page as parameters, the request type and the incoming HTTP headers (cookies, referer, etc.).
- Example Request:
  - http://localhost/example.jsp?
    param1=hello&param2=world

```
<html>
<body>
<p><%= request.getParameter("param1") %></p>
<p><%= request.getParameter("param2") %></p>
</body>
</html>
```



```
<html>
  <body>
    <p>Hello</p>
    <p>World</p>
  </body>
</html>
```

JSP Example: Hello World

1.

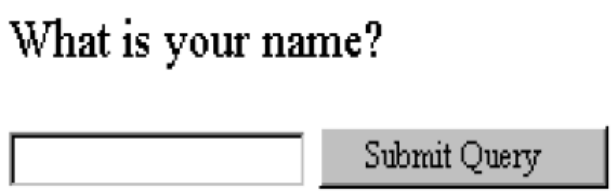
```
<html>
<head>
    <title>Hello World example</title>
</head>
<body>
Hello, World!
</body>
</html>
```

2.

```
<html>
<head>
    <title>Simple JSP Example</title>
</head>

<body>
<FORM METHOD="GET" ACTION="SimpleJSP.jsp">
<p> What is your name?</p>
<INPUT TYPE="TEXT" SIZE="20" NAME="name">
<INPUT TYPE="SUBMIT">
</FORM>

</body>
</html>
```



SimpleJSP.jsp

```
<html>
<head>
    <title>Simple JSP Example - version 1</title>
</head>

<body>
<P>
    <% String visitor = request.getParameter("name");
       if (visitor == null) visitor = "World";   %>
       Hello, <%=visitor%>!<BR>
</P>
</body>
</html>
```

Warning!

- JSP declarations add variables in the servlet instance class
  - Variables shared by all threads (all requests to the same servlet)
  - Until servlet container unloads servlet
  - Beware simultaneous access! Must use synchronized methods

```
<html>
<body>
<%! private int accessCount = 0; %>
   <p> Accesses to page since server reboot:
   <%= ++accessCount %> </p>
</body>
</html>
```

Many HTML Pages are Mostly Static

- Servlets allow us to write dynamic Web pages
  - Easy access to request, session and context data
  - Easy manipulation of the response (cookies, etc.)
  - And lots more...
- It is very inconvenient to write and maintain long and mostly static HTML pages using Servlets (even though such pages are very common)

```
out.println("<h1>Bla Bla</h1>" +  "bla bla bla bla" + "lots
   more here...")
```

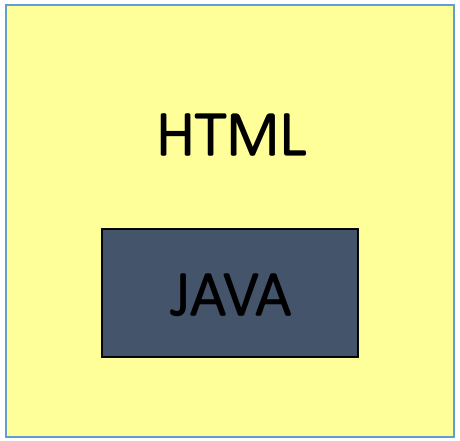
JSP REVIEW

- The Idea:
  - Use HTML for most of the page
  - Write Servlet code directly in the HTML page, marked with special tags
- The server **automatically translates** a JSP page to a **Servlet** class and **invokes** this servlet
  - In Tomcat, you can find the generated Servlet code under <tomcat directory>/work/Catalina/localhost/org/apache/jsp

A JSP is no more than a convenient way to write Servlets that output textual data

Relationships

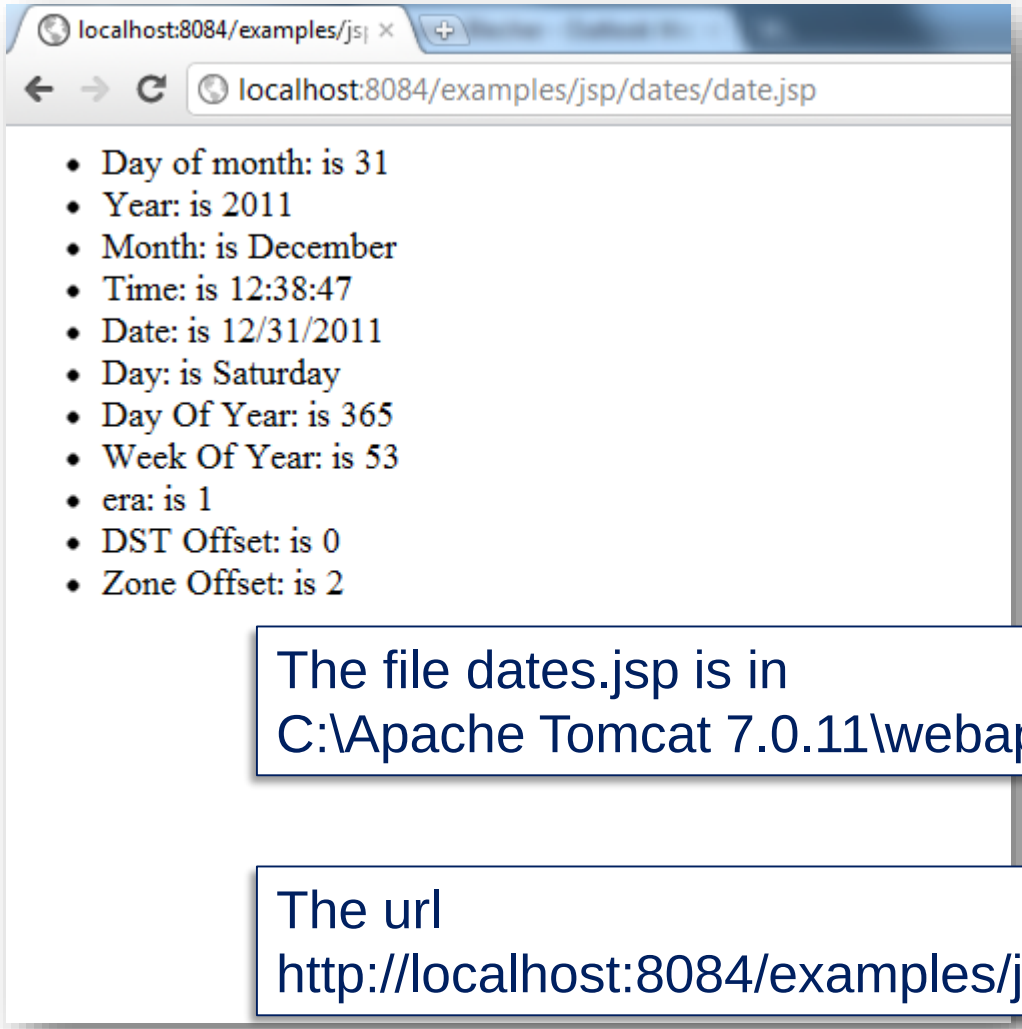
- Servlets: HTML code is printed using Java code
- JSP: Java code is embedded in HTML code
- Not only for HTML!
  - JSP can be used for any textual format
  - Servlets can be used for any data!



JSP – Example

```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <h2><%= new java.util.Date() %></h2>
    <h1>Hello World</h1>
  </body>
</html>
```

JSP – Example (from Tomcat 7)



The file dates.jsp is in
C:\Apache Tomcat 7.0.11\webapps\examples\jsp\dates

The url
http://localhost:8084/examples/jsp/dates/date.jsp

Translation to Servlet

```
package org.apache.jsp.jsp.dates;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class date_jsp extends org.apache.jasper.runtime.HttpJspBase implements
org.apache.jasper.runtime.JspSourceDependent {

  //some more auto generating initialization code

  public void _jspService(final javax.servlet.http.HttpServletRequest request, final
javax.servlet.http.HttpServletResponse response)
      throws java.io.IOException, javax.servlet.ServletException {

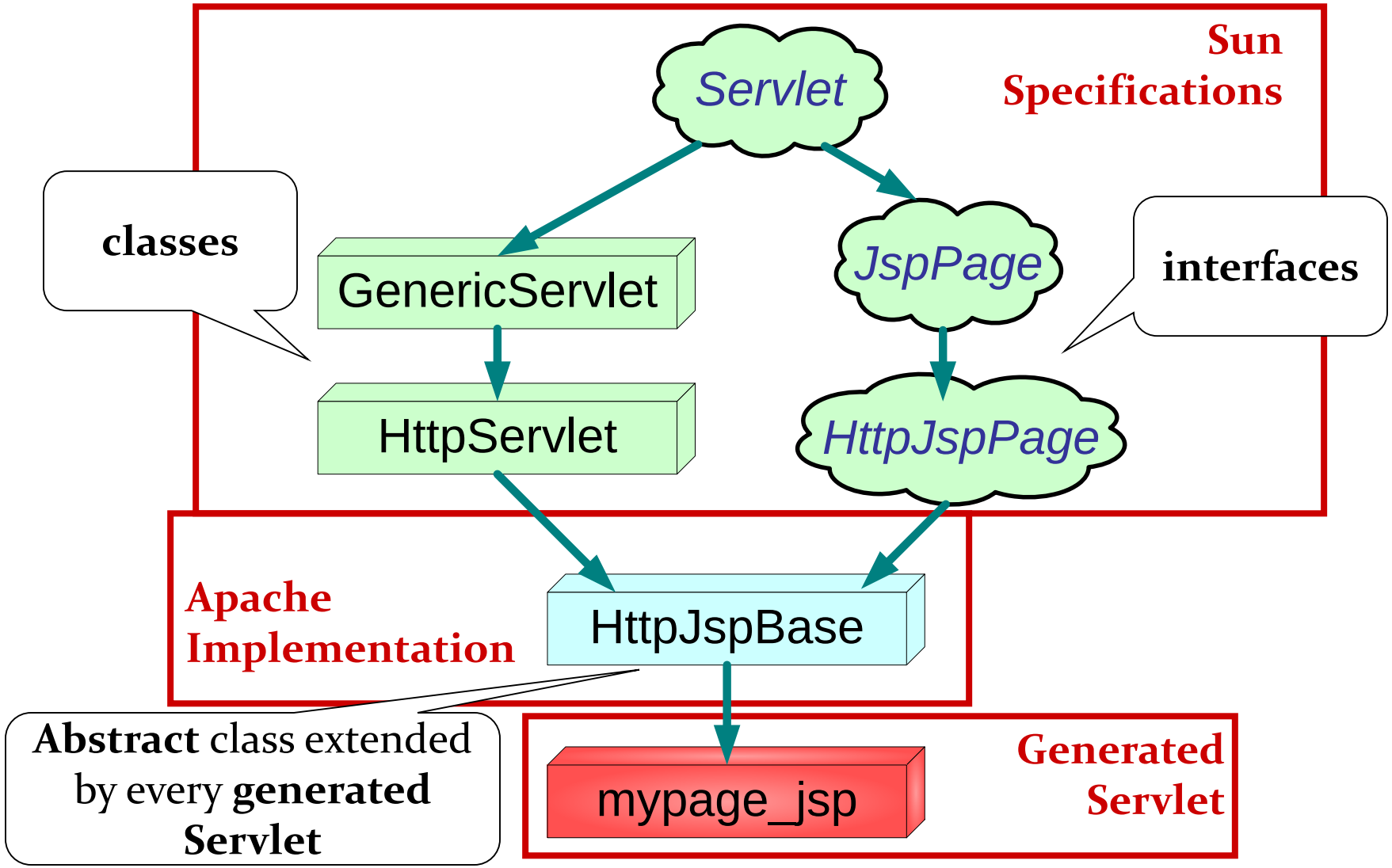
    final javax.servlet.jsp.PageContext pageContext;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;
```

C:\Apache Tomcat 7.0.11\work\Catalina\localhost\examples\org\apache\jsp\jsp\dates\date_jsp.java

```
    try {
      _jspxFactory = JspFactory.getDefaultFactory();
      response.setContentType("text/html");
      pageContext = _jspxFactory.getPageContext(this, request, response,
                                                null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

      out.write("<html>\r\n");
      out.write("  <head>\r\n");
      out.write("    <title>Hello World</title>\r\n");
      out.write("  </head>\r\n");
      out.write("  <body>\r\n");
      out.write("    <h2>");
      out.print( new java.util.Date() );
      out.write("</h2>    \r\n");
      out.write("    <h1>Hello World</h1>\r\n");
      out.write("  </body>\r\n");
      out.write("</html>\r\n");
    } catch (Throwable t) {
      if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          out.clearBuffer();
        if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
      }
    } finally {
      if (_jspxFactory != null) _jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

Generated Servlet Hierarchy (Tomcat Implementation)



JSP Limitations and Advantages

- JSP can only do what a Servlet can do
- Easier to write and maintain HTML
- Easier to separate HTML from code
- Can be created using a "reverse engineering technique":
  - Create static HTML and then replace static data with Java code

JSP Life Cycle

Translation & compilation only after first call...

	Request #1	Request #2	Request #3	Request #4	Request #5	Request #6
JSP page translated into servlet	Yes	No	No	No	Yes	No
JSP's Servlet compiled	Yes	No	No	No	Yes	No
Servlet instantiated and loaded into server's memory	Yes	No	Yes	No	Yes	No
init (or equivalent) called	Yes	No	Yes	No	Yes	No
doGet (or equivalent) called	Yes	Yes	Yes	Yes	Yes	Yes

Page first written

Server restarted

Page modified

JSP Translation

JSP file named **file.jsp** will be translated into the **Java** file **file_jsp.java**

- When the JSP file is modified, it is translated into a Servlet
  - But only **after** the JSP's URL is **requested by a client**
  - The application is not necessarily reloaded immediately when the JSP file is modified
- The server does not generate the Servlet class after startup, if the class already exists and is not too old
- The generated servlet can handle **GET**, **POST**, **HEAD** requests though it does not implement *doGet()*, *doPost()*, *doHead()* explicitly
  - Its *Servlet.service()* method calls the newly implemented main method named *HttpJspBase._jspService()*

init() AND destroy()

- init() of the generated Servlet is called every time the Servlet class is loaded into memory and instantiated
- destroy() of the generated Servlet is called every time the generated Servlet is removed
- init() and destroy() are called even when the reason for loading is a **modification** of the JSP file

jspInit and jspDestroy

- In JSP pages, as in regular Servlets, sometimes we want to implement init and destroy
- It is illegal to use JSP declarations to override init or destroy, since they are (usually) already implemented by the generated Servlet
- Instead, override the methods *jspInit()* and *jspDestroy()*
  - The generated servlet is guaranteed to call these methods from init and destroy, respectively
  - The standard versions of jspInit and jspDestroy are empty (placeholders for you to override)

Thread Synchronization

- After the Servlet is generated, one instance of it serves requests in different threads, just like any other Servlet
- In particular, the service method (_jspService) may be executed by several concurrent threads
- Thus, as with Servlets, JSP programming requires handling concurrency

Basic Elements in a JSP File

- HTML code: <*html-tag*>*content*</*html-tag*>
- JSP Comments: <%-- *comment* --%>
- Expressions: <%= *expression* %>
- Scriptlets (statements): <% *code* %>
- Declarations: <%! *code* %>
- Directives: <%@ *directive attribute*="*value*" %>
- Actions: <jsp:forward.../>, <jsp:include.../>
- Expression-Language Expressions: ${*expression*}

JSP Expressions

- A JSP expression is being used to insert Java values directly into the output
- It has the form: <%= *expression* %> , where *expression* can be a Java object, a numerical expression, a method call that returns a value, etc...
- For example:

```
<%= new java.util.Date() %>
<%= "Hello"+" World" %>
<%= (int)(100*Math.random()) %>
```

The **heading space** and the **following space** are not created in the result.

Use " " if you want a real space

JSP Expressions – Cont.

- Within the generated Java code
  - A JSP Expression is evaluated
  - The result is converted to a string
  - The string is inserted into the page
- This evaluation is performed at runtime (when the page is requested), and thus has full access to information about the request, the session, etc...

Expression Translation

```
<h1>A Random Number</h1>
<%= Math.random() %>
```

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
                        throws java.io.IOException, ServletException {
                         ...
                         response.setContentType("text/html");
                         ...
                         out.write("<h1>A Random Number</h1>\r\n");
                         out.print( Math.random() );
                       out.write("\r\n");
                         ...
}
```

Default content-type

The generated **servlet** calls *out.write()* for **Strings**, and *out.print()* for **objects**

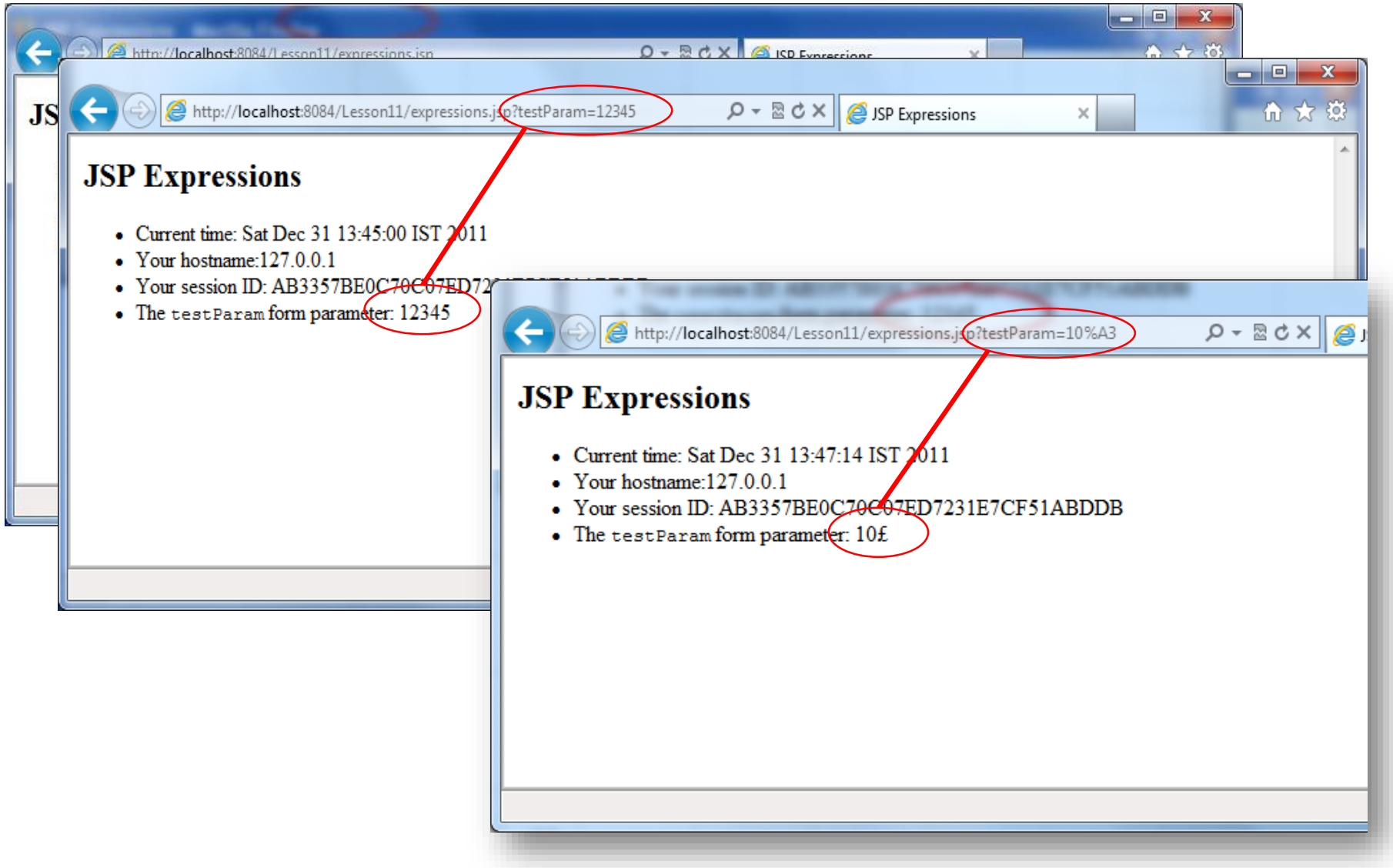
Predefined Variables (Implicit Objects)

- The following predefined variables can be used:
  - **request**: the HttpServletRequest
  - **response**: the HttpServletResponse
  - **session**: the HttpSession associated with the request
  - **out**: the PrintWriter (a buffered version of type JspWriter) used to fill the response content
  - **application**: The ServletContext
  - **config**: The ServletConfig

PREDEFINED VARIABLES - EXAMPLE

```
<html>
 <head>
  <title>JSP Expressions</title>
 </head>
 <body>
  <h2>JSP Expressions</h2>
  <ul>
    <li>Current time: <%= new java.util.Date() %></li>
    <li>Your hostname:<%= request.getRemoteHost() %></li>
    <li>Your session ID: <%= session.getId() %></li>
    <li>The <code>testParam</code> form parameter:
      <%= request.getParameter("testParam") %></li>
  </ul>
 </body>
</html>
```

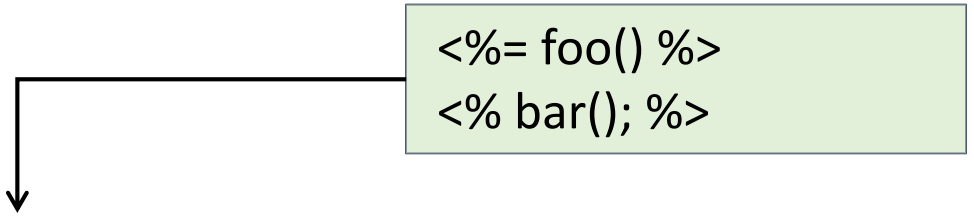
PREDEFINED VARIABLES - EXAMPLE



JSP Scriplets (Statements)

- JSP Scriptlets let you insert arbitrary code into the Servlet service method (_jspService)
- Scriptlets have the form: <% *Java Code* %>
- The code is inserted verbatim into the service method, according to the location of the scriptlet
- Scriptlets have access to the same automatically defined variables as expressions

Scriptlet Translation

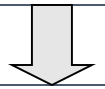



```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
throws ServletException, IOException {
        ...
        response.setContentType("text/html");
        ...
        out.print(foo());
        bar();
        ...
}
```

A Divided-Code Example

- Scriptlets do not have to be continuous

```
<% if (Math.random() < 0.5) { %>
You <b>won</b> the game!
<% } else { %>
You <b>lost</b> the game!
<% } %>
```



```
if (Math.random() < 0.5) {
    out.write("You <b>won</b> the game!");
} else {
    out.write("You <b>lost</b> the game!");
}
```

JSP Declarations

- A JSP declaration lets you define methods or members that are being inserted into the Servlet class (**outside** of all methods)
- It has the following form:

```
<%! Java Code %>
```

- For example:

```
<%! private int someField = 5; %>
<%! private void someMethod(...) {...} %>
```

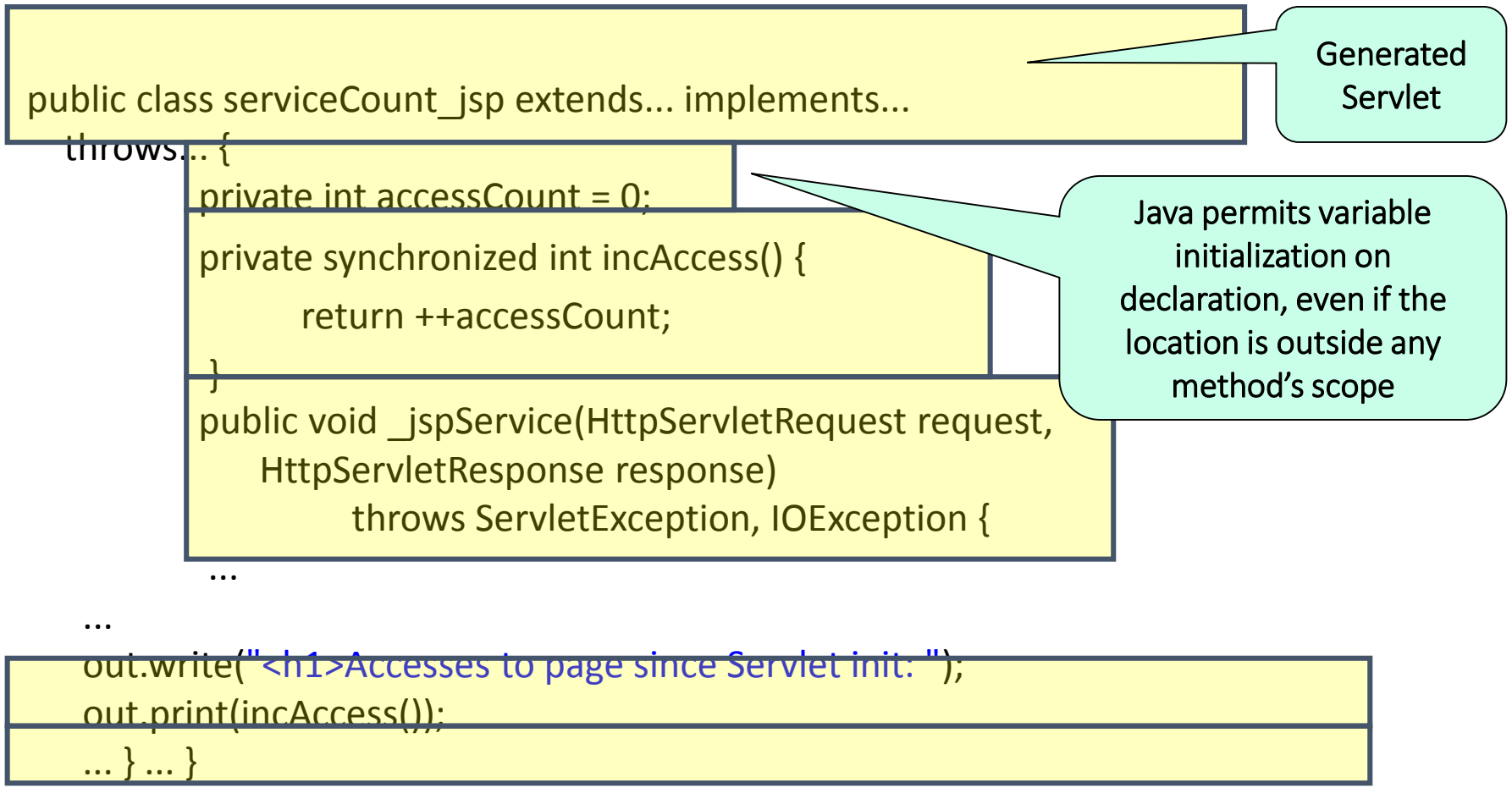
- JSPs are intended to contain a minimal amount of code so it is usually of better design to define methods in a separate Java class...

DECLARATION EXAMPLE

- Print the number of times the current page has been requested since the Servlet initialization:

```
<%! private int accessCount = 0; %>
<%! private synchronized int incAccess() {
                    return ++accessCount;
    } %>
<h1>Accesses to page since Servlet init:
<%= incAccess() %> </h1>
```

Declaration example – cont.



JSP DIRECTIVES

- A JSP directive affects the structure of the Servlet class that is generated from the JSP page
- It usually has the following form:

<%@ *directive attribute1="value1"* **...**
attributeN="valueN" %>

- Three important directives: page, include and taglib

page-Directive Attributes

- import attribute: A comma separated list of classes/packages to import

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

- contentType attribute: Sets the MIME-Type of the resulting document (default is text/html)

```
<%@ page contentType="text/plain" %>
```

page-DIRECTIVE ATTRIBUTES – CONT.

- What is the difference between setting the page contentType attribute, and writing <%response.setContentType("...");%>?
  - In the latter case, the new servlet will call *response.setContentType()* twice
  - The first, impicit (from the JSP point of view), call will be with the default content type.
  - The second, explicit, call might even come after the buffer was flushed or after the writer was obtained…

page-DIRECTIVE ATTRIBUTES – CONT.

- session="**true**|false" specifies if to use a session?
- buffer="*sizekb*|none|**8kb**"
  - Specifies the content-buffer (out) size in kilo-bytes
- autoFlush="**true**|false"
  - Specifies whether the buffer should be flushed when it fills, or throw an exception otherwise
- isELIgnored ="**true**|false"
  - Specifies whether *JSP expression language* is used

The **underlined** values are the defaults

JSP COOPERATION

- We will consider several ways in which JSP and other resources cooperate
  - Forwarding the request handling to other resources
  - Including the content of other sources
  - Including the code of other JSP files
  - Forwarding exception handling to other JSPs

ACTIONS

- JSP ***actions*** use constructs in XML syntax to control the behavior of the Servlet engine
- Using actions, you can
  - **forward** the **request** to another resource in the application
  - dynamically **include** a resource content in the **response**
- Forward and include are translated to an invocation of the RequestDispatcher

The Forward Action

- jsp:forward - Forwards the requester to a new resource

```
<jsp:forward page="{relativeURL|<%= expression %>}">
  <jsp:param name="parameterName"
   value="{parameterValue | <%= expression %>}" /> *
</jsp:forward>
```

0 or more parameters (not attributes!) added to the original request parameters

You can use %=, % instead of <%=, %> so that the code would be a legal XML

Forward Action Example – forward.jsp

```
<%! int even = 0; %>
<% even = (1 - even); %>
<% if (even == 0) { %>
    <jsp:forward page="/requestParams.jsp" >
        <jsp:param name="sessionID" value="<%= session.getId() %>" />
        <jsp:param name="even" value="true" />
    </jsp:forward>
<% } else { %>
    <jsp:forward page="/requestParams.jsp" >
        <jsp:param name="sessionID" value="<%= session.getId() %>" />
        <jsp:param name="even" value="false" />
    </jsp:forward>
<% } %>
```

Forward Action Example – requestParams.jsp

```
<html>
 <head><title>Print Request Params</title></head>
 <body>
  <%@ page import="java.util.*" %>
  <% Enumeration parameterNames = request.getParameterNames(); %>
  <% while (parameterNames.hasMoreElements()) { %>
  <%                  String name = (String)parameterNames.nextElement(); %>
          <h2><%= name %> : <%= request.getParameter(name) %> </h2>
  <% } %>
 </body>
</html>
```

The Include Action

- jsp:include - Include a resource content at **run time**

```
<jsp:include page="{relativeURL|<%= expression %>}">
    <jsp:param name="parameterName"
        value="{parameterValue | <%= expression %>}"
    />*
</jsp:include>
```

0 or more parameters added to the original request parameters

Include Action Example – include.jsp

```
<html>
  <head>
    <title>Include (action) Example</title>
  </head>

  <body>
    <h2>Included part begins:<h2><hr/>
    <jsp:include page="/requestParams2.jsp" >
            <jsp:param name="sessionID" value="<%= session.getId() %>"
    />
    </jsp:include>
    <hr/><h2>Included part ends<h2>
  </body>
</html>
```

Include Action Example – requestParams2.jsp

```
<%@ page import="java.util.*" %>
<% Enumeration parameterNames = request.getParameterNames(); %>
<% while (parameterNames.hasMoreElements()) { %>
<%          String name = (String)parameterNames.nextElement(); %>
<h2><%= name %> : <%= request.getParameter(name) %> </h2>
<% } %>
```

requestParams2.jsp is different from requestParams.jsp in not having the preceding and following html tags (otherwise the output HTML code would have <html>, <head> and <body> duplicated)

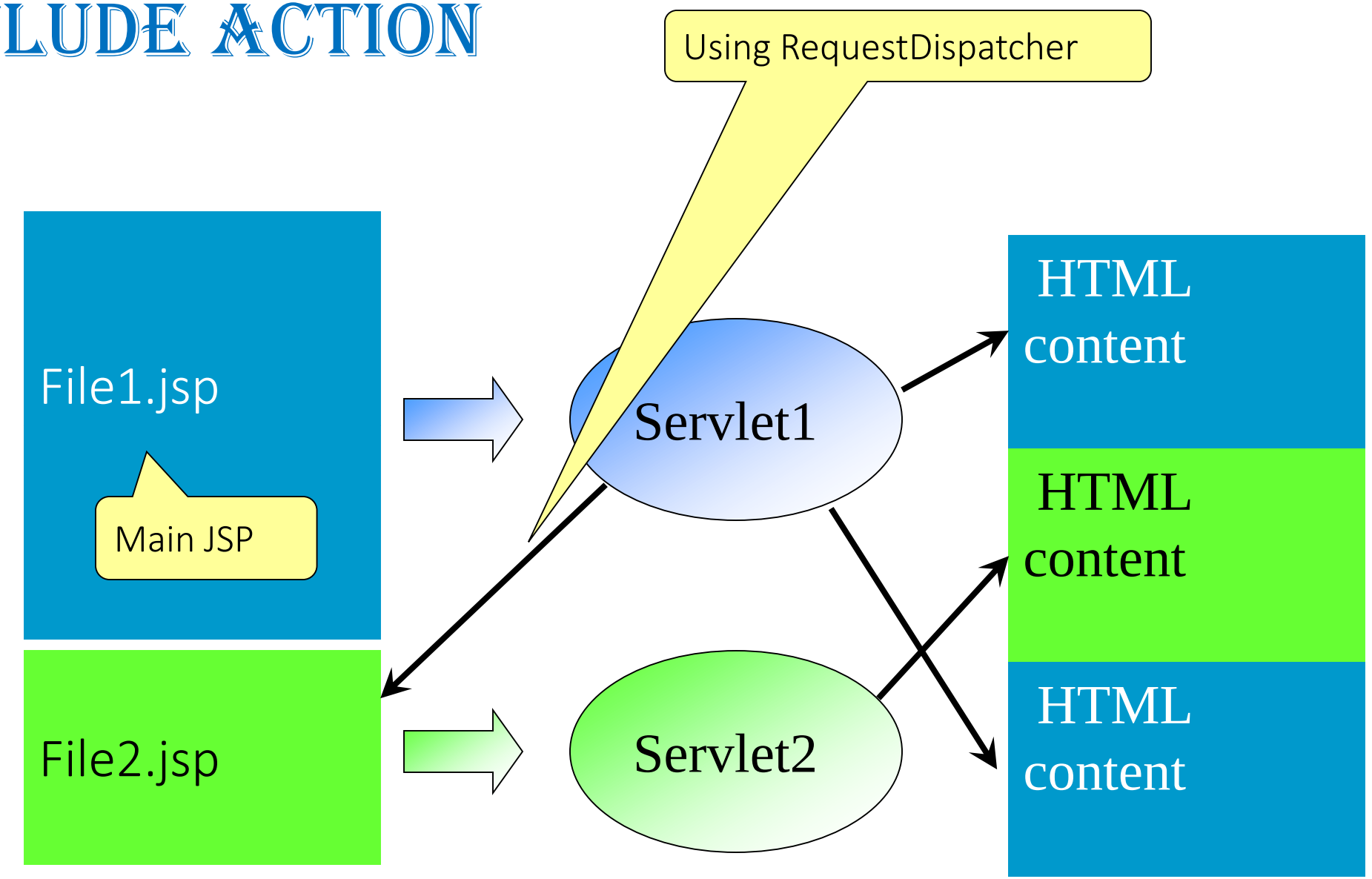
The Include Directive

- This directive lets you include files **at the time the JSP page is translated into a Servlet**
- The directive looks like this:

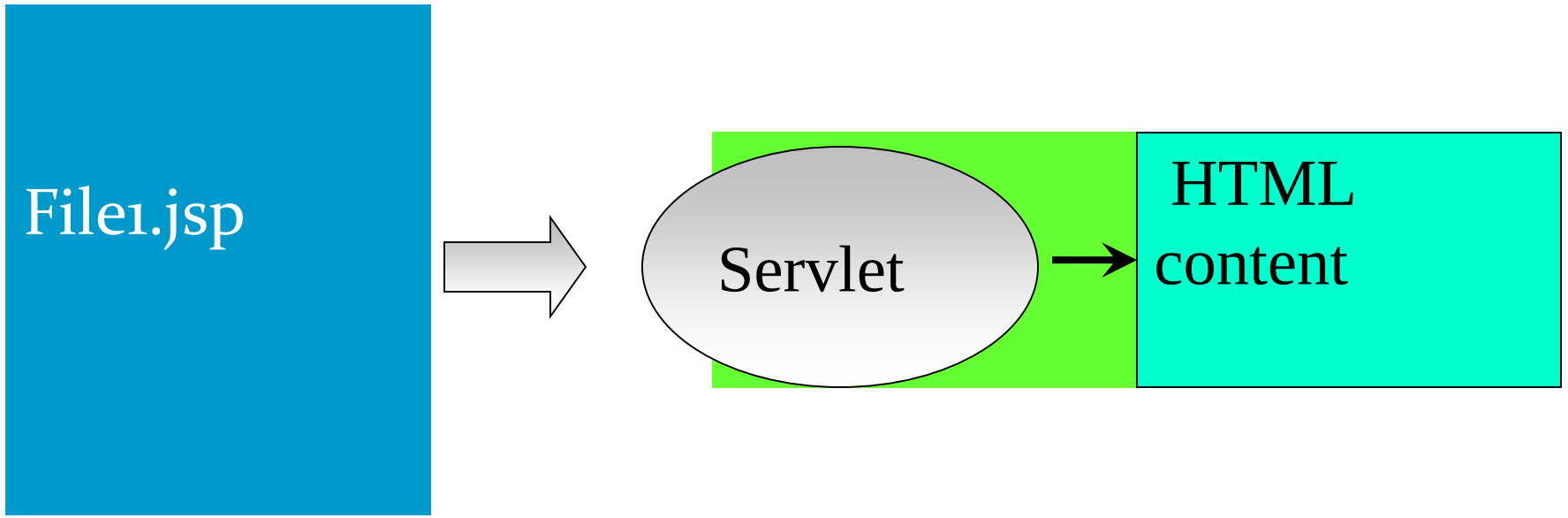
<%@ include file="*url*" %>

- Included JSP content can affect main JSP page
  - e.g. included page directive can affect the result *ContentType*
- Generated Servlets are updated when included files change

INCLUDE ACTION



INCLUDE DIRECTIVE



Include ACTION VS. DIRECTIVE

- When a *resource* is included using the include *action*, the generated Servlet uses the dispatcher to include its content at runtime (so the resource needs not be a JSP or even a Servlet)
- When a *file* is included using the include *directive*, the file itself is included verbatim into the JSP code, prior to the Servlet generation (so the included resource must have JSP syntax)

Include ACTION VS. DIRECTIVE - EXAMPLE

blabla.jsp

```
<html>
 <head><title>Including JSP</title></head><body>
  <h2>Here is an interesting page.</h2>
  <p>Bla, Bla, Bla, Bla.</p>
  <%@ include file="/accessCount.jsp" %>
  <jsp:include page="/myMail.jsp"/>
</body></html>
```

accessCount.jsp

```
<%! private int accessCount = 0; %>
<hr><p>Accesses to page since Servlet init:
<%= ++accessCount %></p>
```

myMail.jsp

```
<hr><p>
Page Created for Simpsons at <%= new java.util.Date() %>. Email <a
href="mailto:homer@springfield.com">here</a>. </p>
```

Include ACTION VS. DIRECTIVE - EXAMPLE

BlaBla_jsp.java

```
out.write("<html>\r\n");
out.write("  <head><title>Including JSP</title></head>\r\n");
out.write("  <body>\r\n");
out.write("    <h2>Here is an interesting page.</h2>\r\n");
out.write("    <p>Bla, Bla, Bla, Bla.</p>\r\n");
```

Original JSP

```
out.write("<hr>\r\n");
out.write("<p> \r\n");
out.write("     Accesses to page since Servlet init: \r\n");
out.print( ++accessCount );
out.write("</p>\r\n");
```

Included JSP

```
org.apache.jasper.runtime.JspRuntimeLibrary.
            include(request, response, "/mymail.jsp", out, false);
```

Similar to RequestDispatcher().include()

```
out.write("  </body>\r\n");
out.write("</html>\r\n");
```

Original JSP

Error Pages

- We can set one JSP page to be the handler of uncaught exceptions of another JSP page, using JSP directives
  - The default behaviour is displa
    500 Internal Server Error with a pa
    stack trace with other exception info
    to the client (ugly and a security risk
  - You can log the entire stack trace alo
    with other data for easier debugging

Runtime exceptions or other exceptions which are declared as thrown by methods your JSP code use. Other exceptions cannot be thrown or else your generated servlet code wouldn't compile

Error Pages – Cont.

- <%@ page errorPage="*url* " %>
  - Defines a JSP page that handles uncaught exceptions
  - The page in *url* should have true in the page-directive:
- <%@ page isErrorPage="true|false" %>
  - The variable exception holds the exception thrown by the calling JSP

Creating an error page without isErrorPage=true, is legal but the exception object is not created in the generated Servlet. If you refer to exception in such a JSP, you'll have a compilation error…

ERROR PAGES - EXAMPLE

connect.jsp

```
<html>
 <head><title>Reading From Database </title></head>
 <body>
  <%@ page import="java.sql.*" %>
  <%@ page errorPage="errorPage.jsp" %>
  <%
   Class.forName(“com.mysql.jdbc.Driver");
   Connection con = DriverManager.getConnection
   (“jdbc:mysql://www.mta.ac.il:3306/homerDB”,
    “homer“, “doughnuts”);
  %>
  <h2>Can Connect!!</h2>
 </body>
</html>
```

Error Pages – Example Cont.

errorPage.jsp

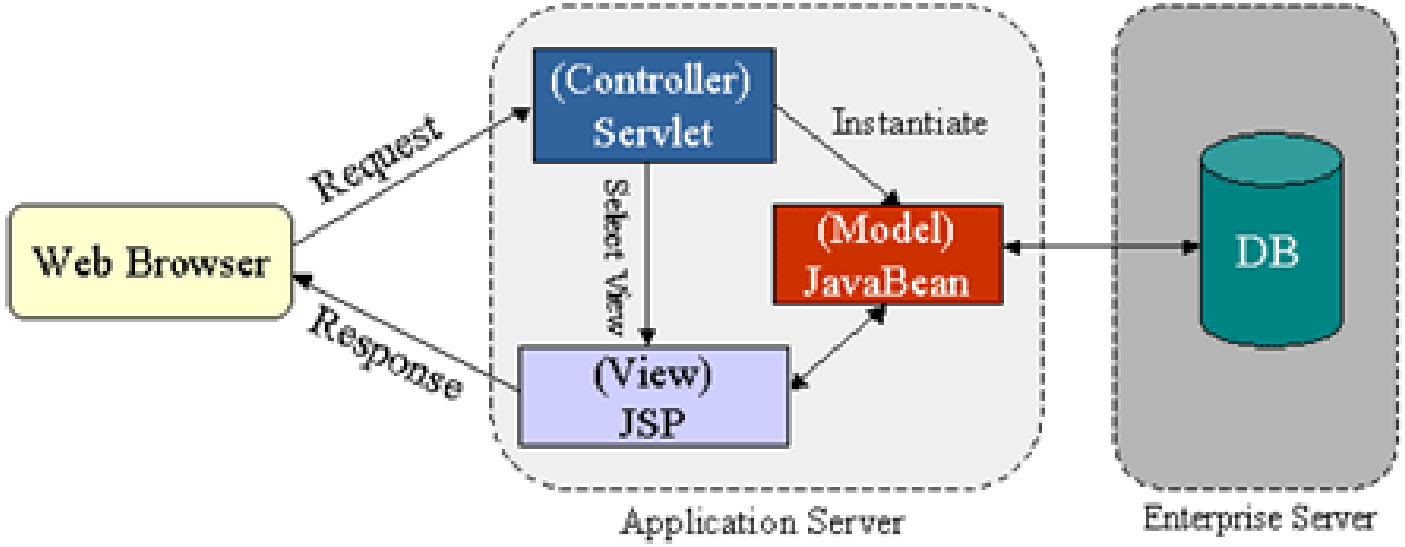
```
<html>
  <head><title>Connection Error</title></head>
  <body>
     <%@ page import="java.io.*" %>
     <%@ page isErrorPage="true" %>
    <h1>Oops. There was an error when you accessed the
        database.</h1>
    <h2>Here is the stack trace:</h2>
    <pre style="color:red">
    <% exception.printStackTrace(new PrintWriter(out)); %>
    </pre>
  </body>
</html>
```

MVC

(model, view, controller)

MVC

- Model
- View
- Controller



The Model

- The Model represents the business logic of the application
- Encapsulating business rules into components:
  - Facilitates testing
  - Improves quality
  - Promotes reuse
- The model can be partitioned into State and Action components

The Model - State Components

- Define the current set of values in the Model and includes methods to update these values.
- Should be protocol independent.
- JavaBeans are a logical choice for implementing State Components.

The Model - Action Components

- Define allowable changes to the State in response to events
- In simpler systems this function may be absorbed into the Controller, however this is not generally recommended.

The View

- Represents the presentation logic of the application.
- Retrieves the State from the Model and provides the user interface for the specific protocol.
- Separating the View from the Model enables the independent construction of user interfaces with different look and feels.
- JSPs are a good choice for implementing the View.

The Controller

- Provides the glue to MVC
- In a MVC system the Controller must handle the following tasks:
  - Security
  - Event Identification
  - Prepare the Model
  - Process the Event
  - Handle Errors
  - Trigger the Response
- Servlets are an ideal choice for the Controller

MVC USING SERVLETS AND JSP

- You can use just JSP to create a web application
- This makes it hard to separate out the web design and the business code
- Using JSP just for the View and Servlets for the Controller can simplify the development process

Servlets and JSP Example: Servlet

- Search Example:

```
ArrayList searchResultsList = // get from the query

RequestDispatcher disp;

disp =
  getServletContext().getRequestDispatcher("searchresults.jsp");

request.setAttribute("my.search.results", searchResultsList);

disp.forward(request, response);
```

- searchResultsList is populated from a query handled by the servlet
- searchresults.jsp is selected as the handler for the response
- We add the search result array to the request so that it is accessible to the JSP

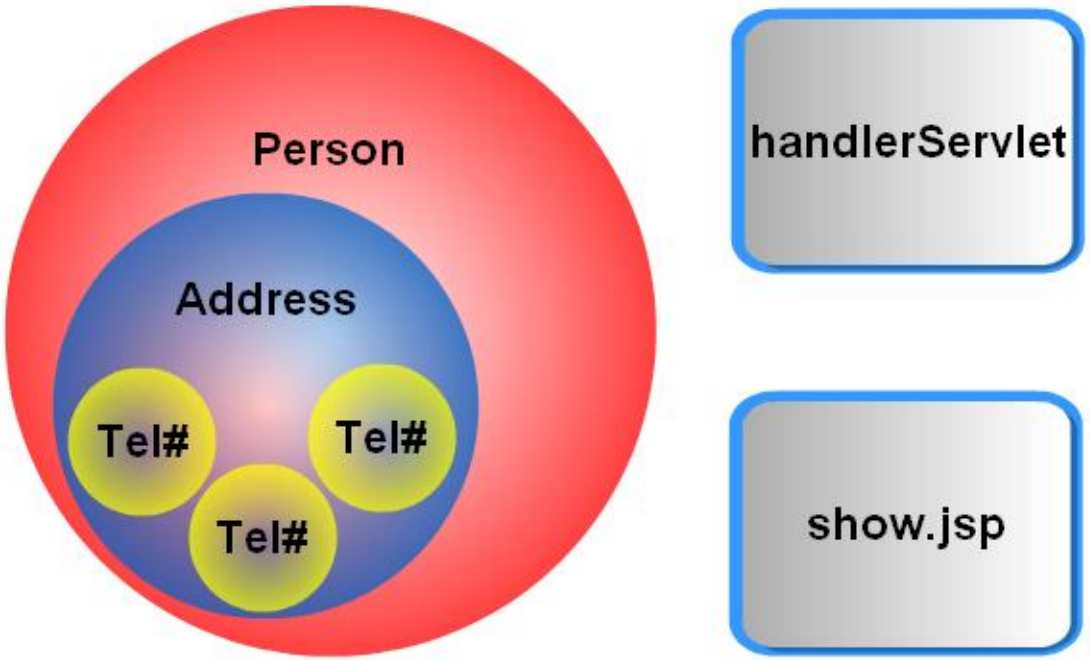
Servlets and JSP Example: JSP

- The JSP can retrieve the result set using this code:

```
ArrayList myList = (ArrayList)
  request.getAttribute("my.search.results");
```

- You can then use a for loop to print the contents of the search request.

Adding JavaBeans Example



Person Bean

```
package webdev.examples.address;


public class Person implements java.io.Serializable {
  private String name;
  private int age;
  private Address address;


  public Person() {
    setName("A N Other");
    setAge(21);
    this.address = new Address();
  }


  public void setName(String name) { this.name = name; }
  public String getName() { return name; }
  public void setAge(int age) { this.age = age; }
  public int getAge() { return age; }
  public void setAddress(Address address) { this.address =
  address; }
  public Address getAddress() { return address; }

}
```

Address Bean

```
package webdev.examples.address;

import java.util.Collection;

public class Address implements java.io.Serializable {
  private String line1;
  private String town;
  private String county;
  private String postcode;
  private Collection phoneNumbers;


  public Address() {
     this.line1 = "line1";
     this.town = "a town2";
     this.county = "a county";
     this.postcode = "postcode";
  }
  public void setLine1(String line1) { this.line1 = line1; }
  public String getLine1() { return line1; }

…
  public Collection getPhoneNumbers() { return phoneNumbers; }
  public void setPhoneNumbers(Collection phoneNumbers) {

      this.phoneNumbers = phoneNumbers;

   }

}
```

PHONENUMBER BEAN

```
package webdev.examples.address;


public class PhoneNumber implements
  java.io.Serializable {
    private String std;
    private String number;


    public String getNumber() { return number; }
    public String getStd() { return std; }
    public void setNumber(String number) {
      this.number = number; }
   public void setStd(String std) {
      this.std = std; }
}
```

HANDLERSERVLET.JAVA

```
package webdev.examples.address;

import java.io.IOException; import java.util.ArrayList; import javax.servlet.RequestDispatcher;

import javax.servlet.ServletException; import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest; import javax.servlet.http.HttpServletResponse;


public class handlerServlet extends HttpServlet {

protected void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException,IOException

  {
   Person p = new Person();
   p.setName("Sam Dalton");
   p.setAge(26);
   Address a = new Address();
   a.setLine1("221b Baker Street");
   a.setTown("London");
   a.setCounty("Greater London");
   a.setPostcode("NW1 1AA");
   ArrayList al = new ArrayList();
   PhoneNumber ph = new PhoneNumber();
   ph.setStd("01895");
   ph.setStd("678901");
   al.add(ph);
   ph = new PhoneNumber();
   ph.setStd("0208");
   ph.setStd("8654789");
   al.add(ph);
   a.setPhoneNumbers(al);
   p.setAddress(a);
   req.setAttribute("person", p);
   RequestDispatcher rd = req.getRequestDispatcher("show.jsp");
   rd.forward(req, res);

  }

}
```

Setting the Request Variable

```
req.setAttribute("person", p);
RequestDispatcher rd =
  req.getRequestDispatcher(“show.jsp");
rd.forward(req, res);
```

SHOW.JSP

```
<html>
  <head>
  <title>MVC Example</title>
  </head>
  <body>
  <h2>MVC Example</h2>
  <table border="1">
  <tr>
  <td>${person.name}</td>
  <td>${person.age}</td>
  <td>${person["address"].line1}</td>
  <td>${person["address"].town}</td>
  <td>${person.address.phoneNumbers[0].std}
  ${person.address.phoneNumbers[0].number}</td>
  <td>${person.address.phoneNumbers[1].std}
  ${person.address.phoneNumbers[1].number}</td>
  </tr>
  </table>
  </body>
  </html>
```

OUTPUT


EL and Complex JavaBeans - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/expressionLanguage/BeanTest
Go Links
EL and Complex JavaBeans
Sam Dalton 26 221b Baker Street London 678901 8654789
Done
Local intranet

3) Spring Framework

What is Spring Framework?

- Spring framework is an open source Java platform that provides comprehensive infrastructure support for developing robust Java applications very easily and very rapidly. Spring framework was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

- The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use

WHY SPRING?...

- Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
- Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.

WHY SPRING?

- Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.

DEPENDENCY INJECTION (DI)...

- The technology that Spring is most identified with is the Dependency Injection (DI) flavor of Inversion of Control. The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways. Dependency Injection is merely one concrete example of Inversion of Control.

- When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection helps in gluing these classes together and at the same time keeping them independent.

DEPENDENCY INJECTION (DI)

- The dependency part translates into an association between two classes. For example, class A is dependent of class B. Now, let's look at the second part, injection. All this means is, class B will get injected into class A by the IoC.

- Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods.

AOP (ASPECT ORIENTED PROGRAMING)

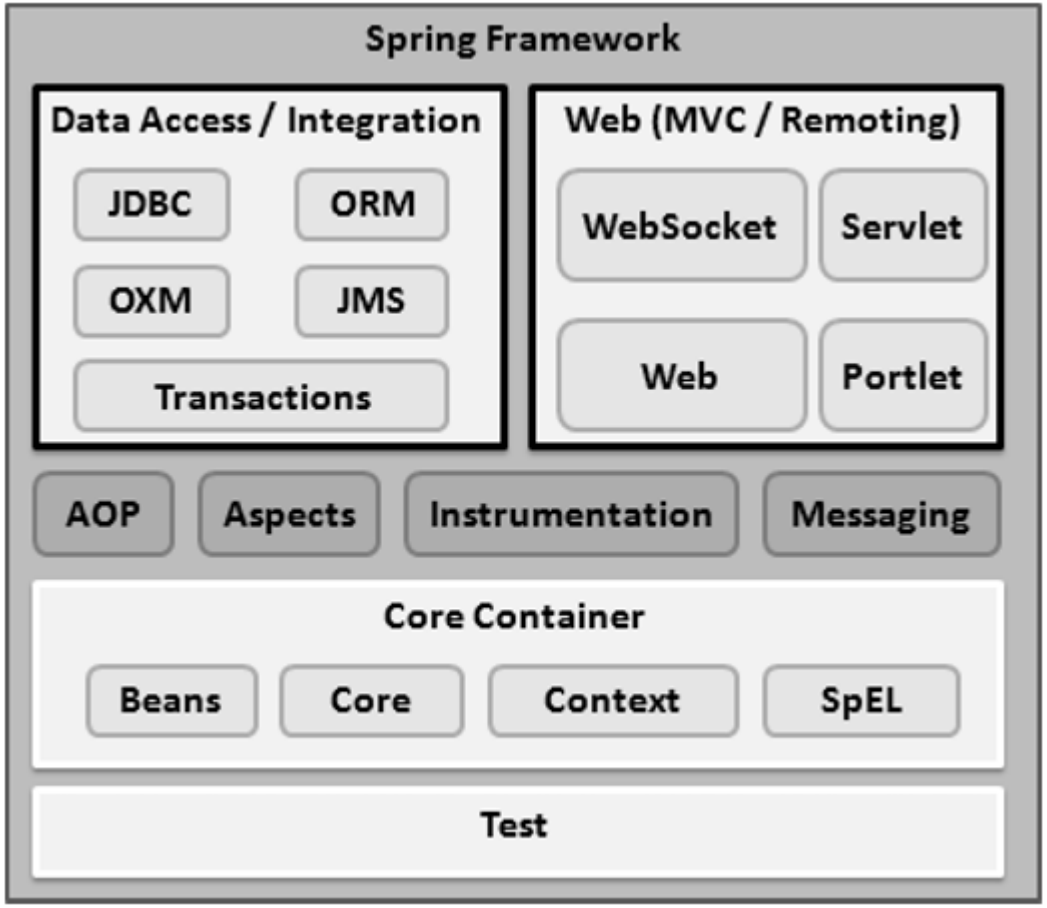
- (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.

Spring Framework – Architecture…

- Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest. The following section provides details about all the modules available in Spring Framework.

- The Spring Framework provides about 20 modules which can be used based on an application requirement.

Spring Framework – Architecture...



CORE CONTAINER

- The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows −

- The Core module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.

- The Bean module provides BeanFactory, which is a sophisticated implementation of the factory pattern.

- The Context module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.

- The SpEL module provides a powerful expression language for querying and manipulating an object graph at runtime.

DATA ACCESS/INTEGRATION

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −

- The JDBC module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
- The ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.

- The Java Messaging Service JMS module contains features for producing and consuming messages.
- The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

THE WEB LAYER

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows −

- The Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.

- The Web-MVC module contains Spring's Model-View-Controller (MVC) implementation for web applications.

- The Web-Socket module provides support for WebSocket-based, two-way communication between the client and the server in web applications.

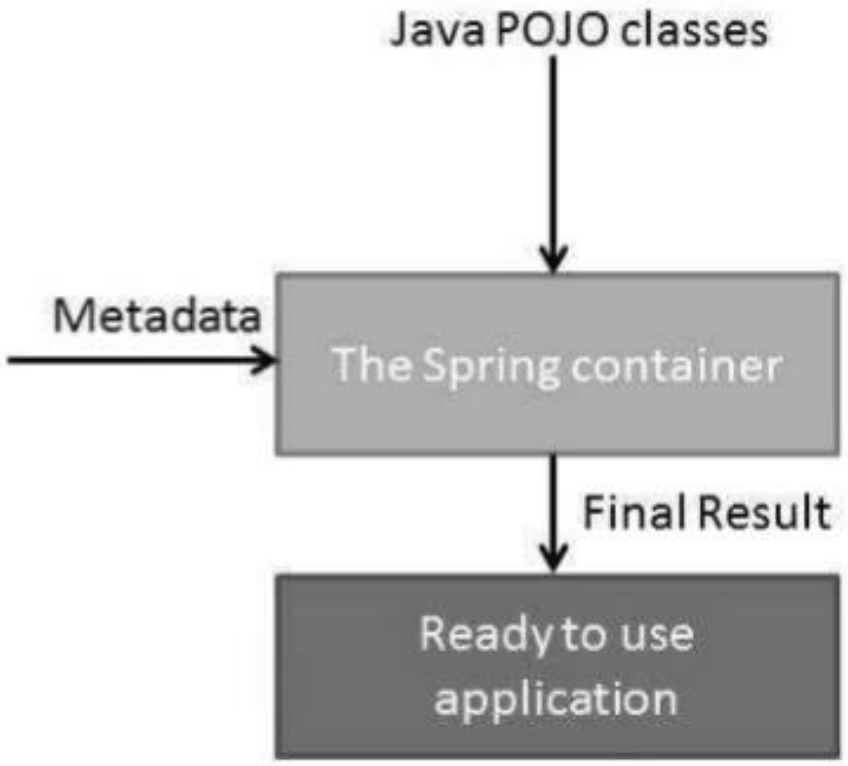
- The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

SPRING - IOC CONTAINERS...

- The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans.

- The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

SPRING - IOC CONTAINERS...



Spring BeanFactory Container:

This is the simplest container providing the basic support for DI and is defined by the org.springframework.beans.factory.BeanFactory interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring.

Spring ApplicationContext Container:

This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

SPRING - BEAN DEFINITION

The objects that form the backbone of your application and that are managed by the Spring IoC container are called **beans**. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that we supply to the container.

Bean definition contains the information called **configuration metadata**, which is needed for the container to know the following −

- How to create a bean
- Bean's lifecycle details
- Bean's dependencies

SPRING - DEPENDENCY INJECTION...

```
public class TextEditor {
    private SpellChecker spellChecker;

    public TextEditor() {
        spellChecker = new SpellChecker();
    }
}
```

```
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker
spellChecker) {
      this.spellChecker = spellChecker;
   }
}
```

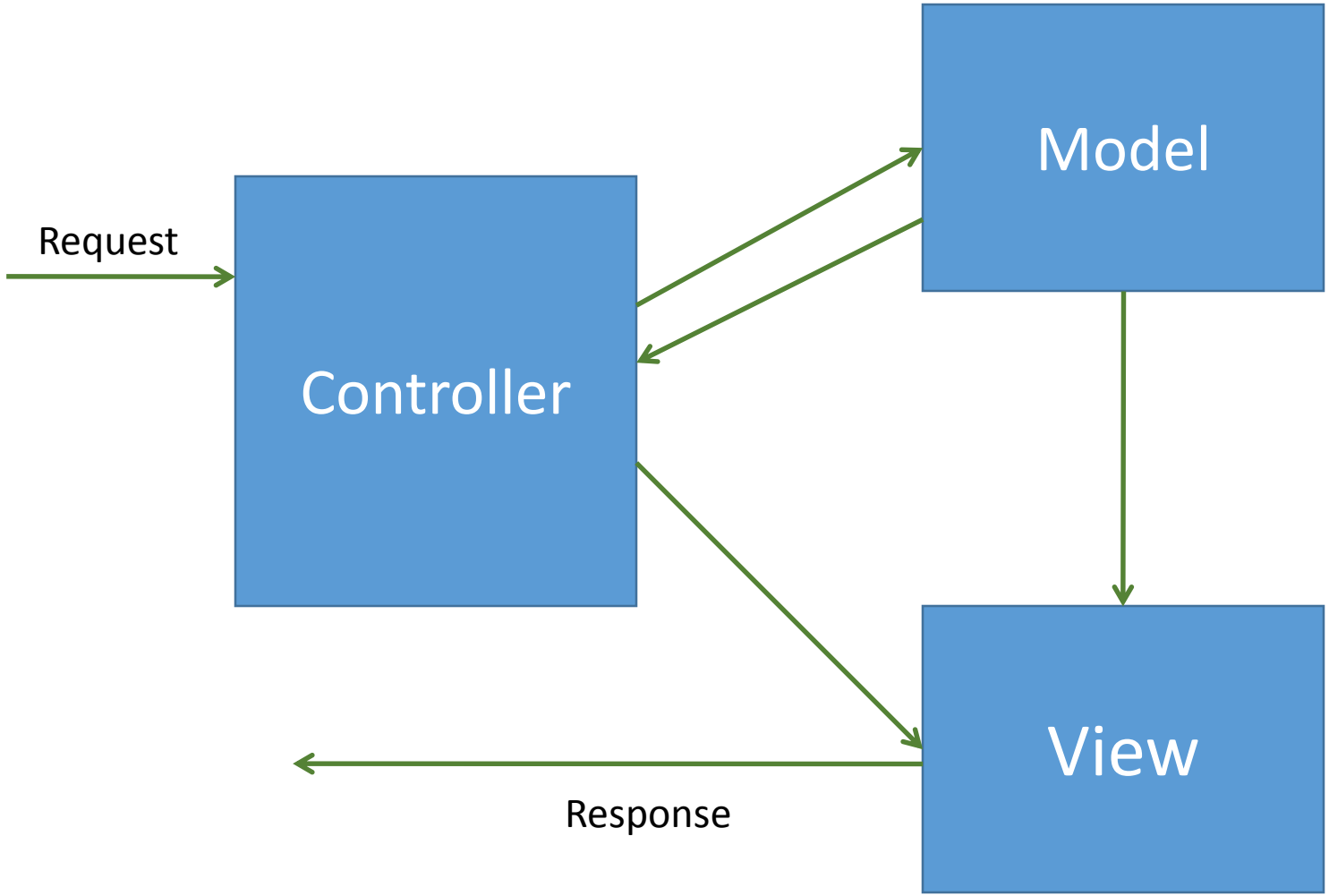
SPRING - DEPENDENCY INJECTION

- Every Java-based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent.

Advantages of Spring 3.0 MVC

- Annotation based configuration.
- Supports to plug with other MVC frameworks like Struts etc.
- Flexible in supporting different view types like JSP, velocity, XML, PDF etc.,

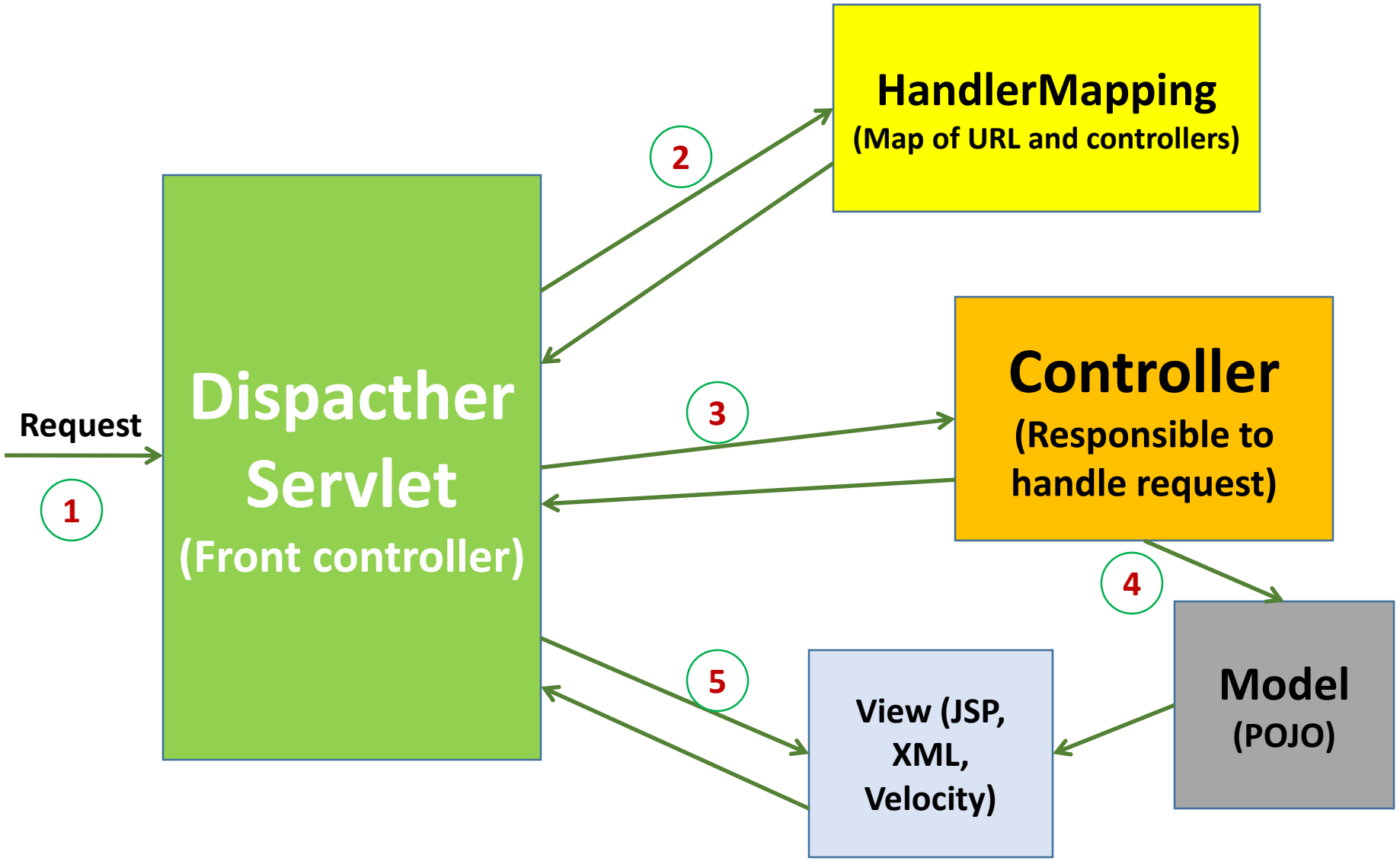
MVC – AN OVERVIEW



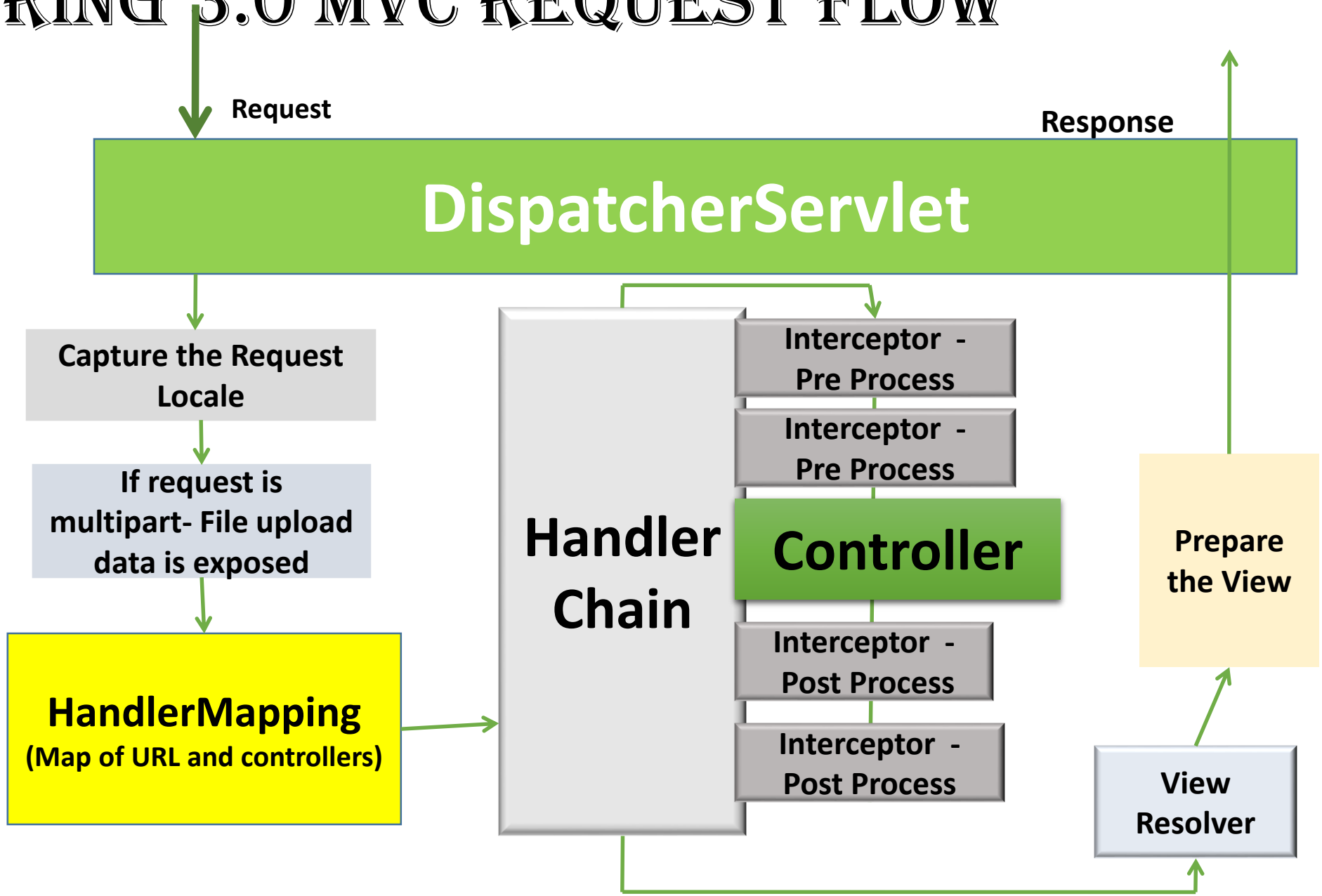
Controller - Responsibilities

- Initialize the framework to cater to the requests.
- Load the map of all the URLs and the components responsible to handle the request.
- Prepare the map for the views.

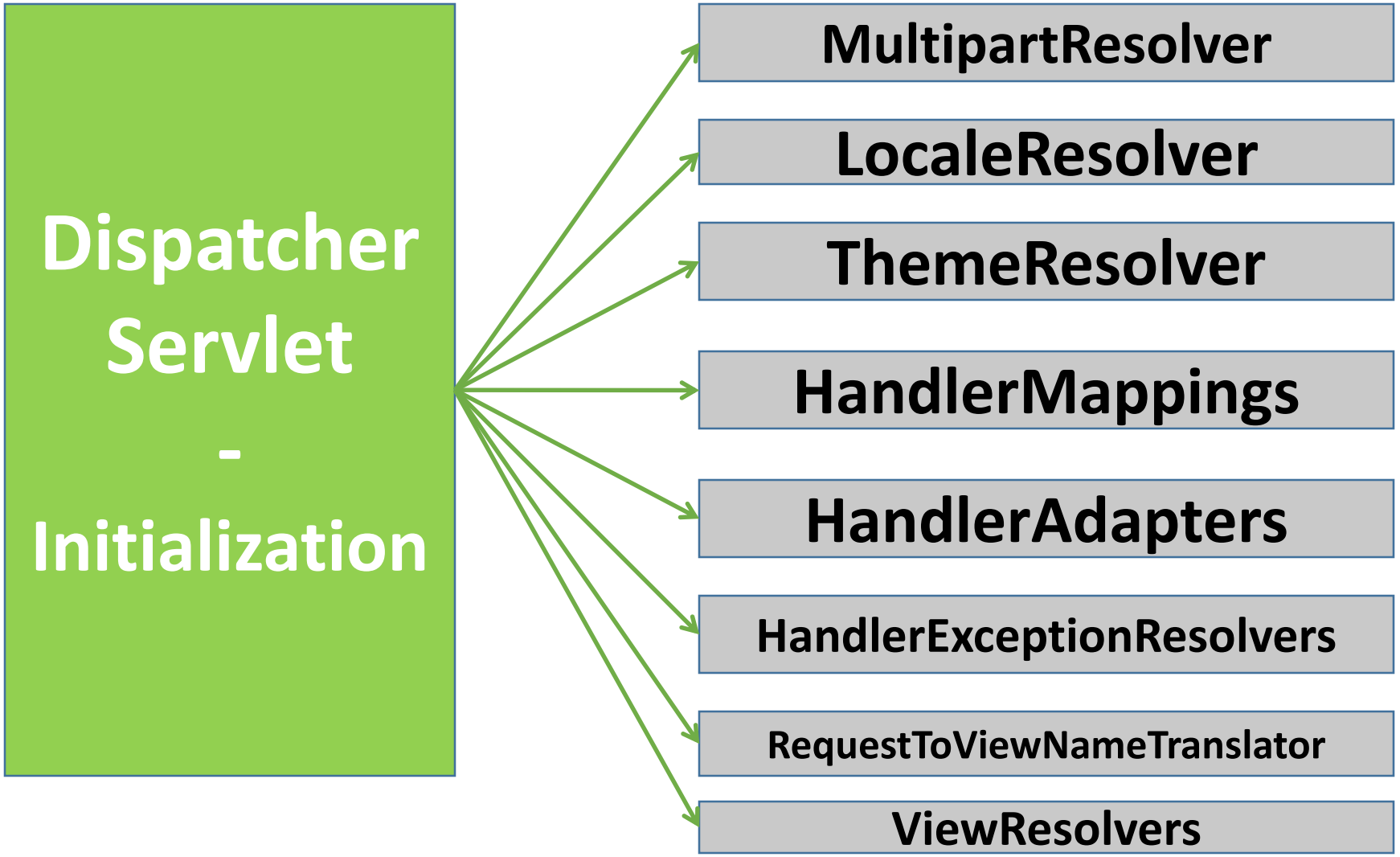
SPRING 3 MVC- BASIC ARCHITECTURE



SPRING 3.0 MVC REQUEST FLOW



SPRING 3 MVC FRAMEWORK-INITIALIZATION

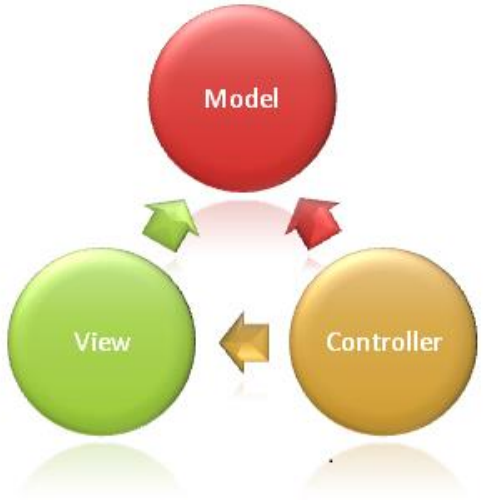


IMPORTANT INTERFACES

Interface	Default bean name	purpose
org.springframework.web.servlet.HandlerMapping	handlerMapping	Maps the Request to Handlers(Controllers)
org.springframework.web.servlet.HandlerAdapter	none	Plugs the other frameworks handlers
org.springframework.web.servlet.ViewResolver	viewResolver	Maps the view names to view instances
org.springframework.web.servlet.HandlerExceptionResolver	handlerExceptionResolver	Mapping of the exceptions to handlers and views
org.springframework.web.multipart.MultipartResolver	multipartResolver	Interface to handle the file uploads
org.springframework.web.servlet.LocaleResolver	localeResolver	Helps to resolve the locale from the request
org.springframework.web.servlet.ThemeResolver	themeResolver	Resolves a theme for a Request.

MVC Review

- MVC is a approach to building complex applications that breaks the design up into three components: The Model, the View and the Controller:



The MVC Pattern

- Model-View-Controller ("MVC") an architectural design pattern for interactive applications dividing tasks into three separate modules:
  - one for the application model with its data representation and business logic,
  - the second for views that provide data presentation and user input, and
  - the third for a controller to dispatch requests and control flow.

The MVC Pattern

- In the MVC Design Pattern:
  - The **view** manages the graphical and/or textual output to the portion of the interaction with the user.
  - The **controller** interprets the inputs from the user, commanding the model and/or the view to change as appropriate.
  - The **model** manages the behavior of the data and the state of the application domain.

The MVC Pattern

- Most Web-tier application frameworks use some variation of the MVC design pattern
- The MVC (architectual) design pattern provides a host of design benefits

Benefits of MVC

- Clarity of design
  - easier to implement and maintain
- Modularity
  - changes to one don't affect the others
  - can develop in parallel once you have the interfaces
- Supports Multiple domains
  - Web, desktops, games, spreadsheets, Powerpoint, IDEs, UML reverse engineering, ….

MODEL

- The Model's responsibilities
  - Provide access to the state of the system
  - Provide access to the system's functionality
  - Can notify the view(s) that its state has changed

View

- The view's responsibilities
  - Display the state of the model to the user
- At some point, the model (a.k.a. the observable) must registers the views (a.k.a. observers) so the model can notify the observers that its state has changed

Controller

- The controller's responsibilities
  - Accept user input
    - Button clicks, key presses, mouse movements, slider bar changes
  - Send information to the model
  - Send appropriate information to the view

MVC in a Web Interaction