

# **INTRODUCTION TO JAVASCRIPT**

# JAVA SCRIPT

## Introduction

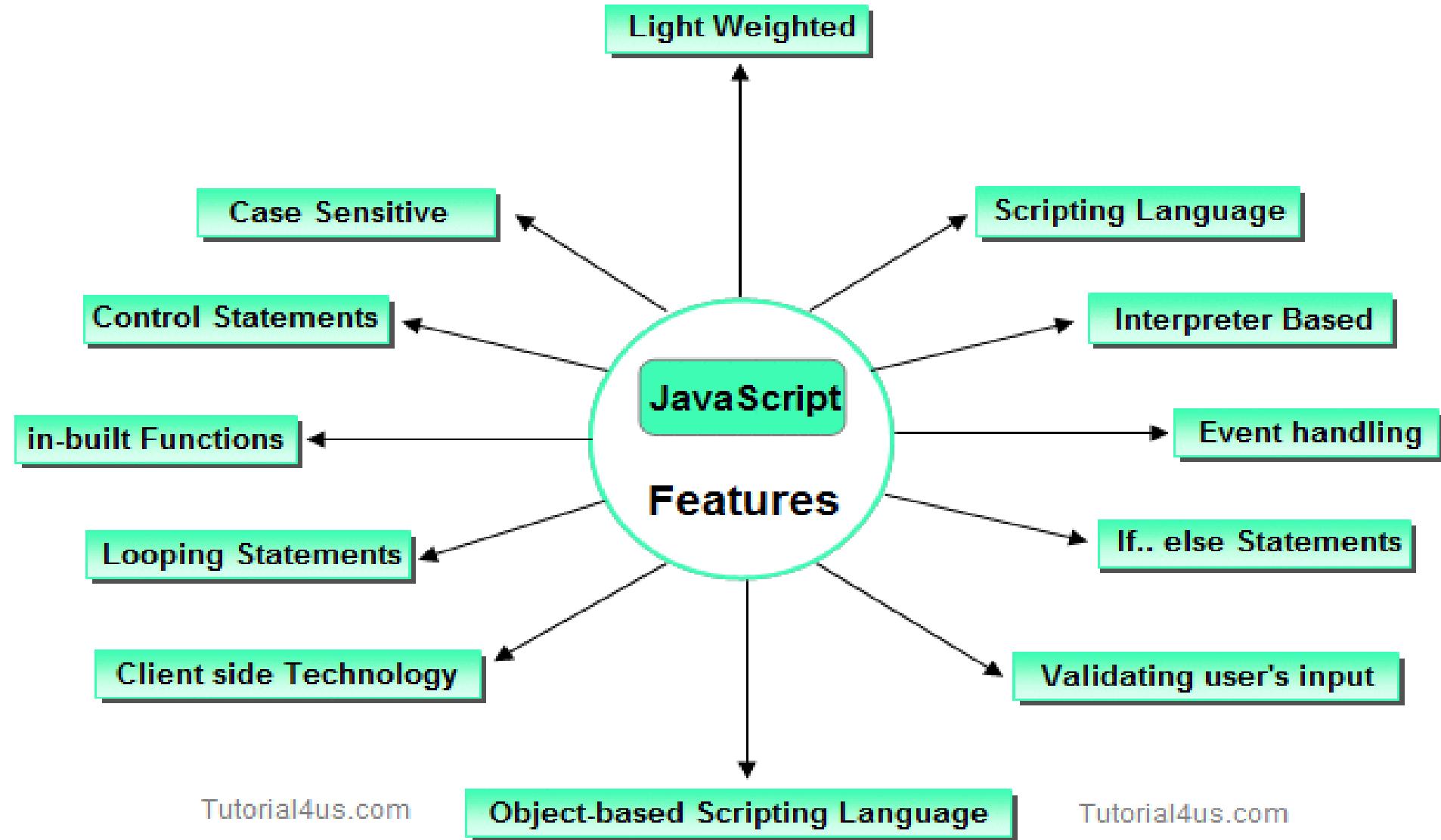
- **JavaScript** is a object-based scripting language and it is light weighted.
- It is first implemented by Netscape (with help from Sun Microsystems).
- JavaScript was created by **Brendan Eich** at Netscape in 1995 for the purpose of allowing code in web-pages (performing logical operation on client side).
- It is not compiled but translated. JavaScript Translator is responsible to translate the JavaScript code which is embedded in browser.

# Where it is used?

- It is used to create interactive websites. It is mainly used for:
- Client-side validation
- Dynamic drop-down menus
- Displaying date and time
- Build small but complete client side programs .
- Displaying popup windows and dialog boxes (like alert dialog box, confirm dialog box and prompt dialog box)
- Displaying clocks etc.

# Features of JavaScript

JavaScript is a client side technology, it is mainly used for gives client side validation, but it have lot of features which are given below;



# Way of Using JavaScript

- There are three places to put the JavaScript code.
- Between the <body> </body> tag of html  
(Inline JavaScript)
- Between the <head> </head> tag of html  
(Internal JavaScript)
- In .js file (External JavaScript)

# How to Put a JavaScript Into an HTML Page?

```
<html>
<body>
<script type="text/javascript">
document.write("Hello World!")
</script>
</body>
</html>
```

- **Inline JavaScript**
- When java script was written within the html element using attributes related to events of the element then it is called as inline java script.
- **Example of Inline JavaScript**
- **Example How to use JavaScript**
- **<html> <form> <input type="button" value="Click" onclick="alert('Button Clicked')"/> </form> </html>**
- Output:

Click

# Internal JavaScript

- When java script was written within the section using element then it is called as internal java script.
- **Example of Internal JavaScript**
- <html>
- <head>
- <script>
- **function msg()**
- { alert("Welcome in JavaScript"); }
- </script>
- </head>
- <form>
- <input type="button" value="Click" onclick="msg()"/>
- </form>
- </html>
- Output:

click

# External JavaScript

- Writing java script in a separate file with extension .js is called as external java script. For adding the reference of an external java script file to your html page, use tag with src attribute as follows
- **Example**
- **<script type="text/javascript" src="filename.js"/>**
- Create a file with name functions.js and write the following java script functions in it.
- **message.js**
- **Example**
- **function msg() { alert("Welcome in JavaScript"); }**

- Example
- <html>
- <head>
- <script type="text/javascript" src="message.js">
- </script>
- </head>
- <body>
- <form>
- <input type="button" value="click" onclick="msg()" />
- </form>
- </body>
- </html>
- output:

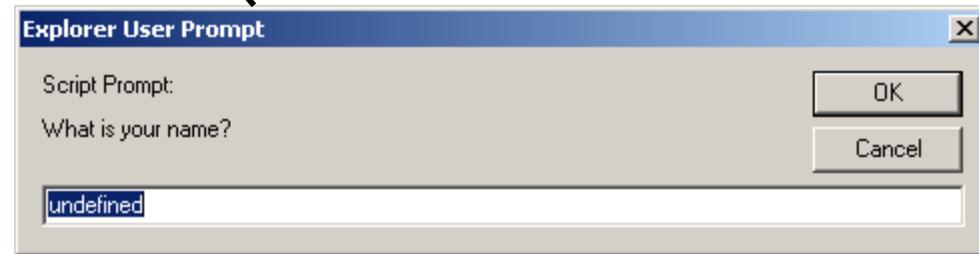
click

# Hiding JavaScript from Incompatible Browsers

```
<script type="text/javascript">  
  <!--  
    document.writeln("Hello, WWW");  
  // -->  
</script>  
  
<noscript>  
  Your browser does not support JavaScript.  
</noscript>
```

# **alert(), confirm(), and prompt()**

```
<script type="text/javascript">  
alert("This is an Alert method");  
confirm("Are you OK?");  
prompt("What is your name?");  
prompt("How old are you?", "20");  
</script>
```



# **alert() and confirm()**

```
alert("Text to be displayed");
```

- Display a message in a dialog box.
- The dialog box will block the browser.

```
var answer = confirm("Are you sure?");
```

- Display a message in a dialog box with two buttons: "OK" or "Cancel".
- confirm() returns `true` if the user click "OK". Otherwise it returns `false`.

# **prompt()**

```
prompt("What is your student id number?");  
prompt("What is your name?", "No name");
```

- Display a message and allow the user to enter a value
- The second argument is the "default value" to be displayed in the input textfield.
- Without the default value, "undefined" is shown in the input textfield.
- If the user click the "OK" button, **prompt()** returns the value in the input textfield as a string.
- If the user click the "Cancel" button, **prompt()** returns null.

# Identifier

- Same as Java/C++ except that it allows an additional character – '\$'.
- Contains only 'A' – 'Z', 'a' – 'z', '0' – '9', '\_', '\$'
- First character cannot be a digit
- Case-sensitive
- Cannot be reserved words or keywords

# Variable and Variable Declaration

```
<head><script type="text/javascript">
    // We are in the default scope - the "window" object.
    x = 3;          // same as "window.x = 3"
    var y = 4;      // same as "y = 4" or "window.y = 4"

    {   // Introduce a block to create a local scope
        x = 0;          // Same as "window.x = 0"
        var y = 1;      // This is a local variable y
    }

    alert("x=" + x + ", y=" + y); // Print x=0, y=4
</script></head>
```

- Local variable is declared using the keyword 'var'.
- Dynamic binding – a variable can hold any type of value
- If a variable is used without being declared, the variable is created automatically.
  - If you misspell a variable name, program will still run (but works incorrectly)

# Data Types

- Primitive data types
  - Number: integer & floating-point numbers
  - Boolean: true or false
  - String: a sequence of alphanumeric characters
- Composite data types (or Complex data types)
  - Object: a named collection of data
  - Array: a sequence of values (an array is actually a predefined object)
- Special data types
  - Null: the only value is "null" – to represent nothing.
  - Undefined: the only value is "undefined" – to represent the value of an uninitialized variable

# Strings

- A string variable can store a sequence of alphanumeric characters, spaces and special characters.
- Each character is represented using 16 bit
  - You can store Chinese characters in a string.
- A string can be enclosed by a pair of single quotes (') or double quote (").
- Use escaped character sequence to represent special character (e.g.: \" , \n , \t)

# typeof operator

```
var x = "hello", y;  
alert("Variable x value is " + typeof x );  
alert("Variable y value is " + typeof y );  
alert("Variable x value is " + typeof z );
```

- An unary operator that tells the type of its operand.
  - Returns a string which can be "number", "string", "boolean", "object", "function", "undefined", and "null"
  - An array is internally represented as an object.

# Operators

- Arithmetic operators
  - +, -, \*, /, %
- Post/pre increment/decrement
  - ++, --
- Comparison operators
  - ==, !=, >, >=, <, <=
  - ===, !== (Strictly equals and strictly not equals)
    - i.e., Type and value of operand must match / must not match

## **== VS ===**

```
// Type conversion is performed before comparison
var v1 = ("5" == 5);      // true

// No implicit type conversion.
// True if only if both types and values are equal
var v2 = ("5" === 5);    // false

var v3 = (5 === 5.0);   // true

var v4 = (true == 1);   // true (true is converted to 1)

var v5 = (true == 2);   // false (true is converted to 1)

var v6 = (true == "1") // true
```

# Logical Operators

- **!** – Logical NOT
- **&&** – Logical AND
  - OP1 && OP2
  - If OP1 is true, expression evaluates to the value of OP2.  
Otherwise the expression evaluates to the value of OP1.
  - Results may not be a boolean value.
- **||** – Logical OR
  - OP1 || OP2
  - If OP1 is true, expression evaluates to the value of OP1.  
Otherwise the expression evaluates to the value of OP2.

```
var tmp1 = null && 1000;           // tmp1 is null

var tmp2 = 1000 && 500;           // tmp2 is 500

var tmp3 = false || 500;          // tmp3 is 500

var tmp4 = "" || null;           // tmp4 is null

var tmp5 = 1000 || false;         // tmp5 is 1000

// If foo is null, undefined, false, zero, NaN,
// or an empty string, then set foo to 100.
foo = foo || 100;
```

# Operators (continue)

- String concatenation operator
  - +
  - If one of the operand is a string, the other operand is automatically converted to its equivalent string value.
- Assignment operators
  - =, +=, -=, \*=, /=, %=
- Bitwise operators
  - &, |, ^, >>, <<, >>>

# Conditional Statements

- “if” statement
- “if ... else” statement
- "? :" ternary conditional statement
- “switch” statement
  
- The syntax of these statements are similar to those found in C and Java.

# Conditional Statements

In JavaScript we have the following conditional statements:

- **if statement** - use this statement if you want to execute some code only if a specified condition is true
- **if...else statement** - use this statement if you want to execute some code if the condition is true and another code if the condition is false
- **if...else if....else statement** - use this statement if you want to select one of many blocks of code to be executed
- **switch statement** - use this statement if you want to select one of many blocks of code to be executed

# Conditional Statements - 2

```
if (condition)
{
code to be executed if condition is true
}
```

---

```
if (condition)
{
code to be executed if condition is true
}
else
{
code to be executed if condition is not true
}
```

# Conditional Statements Examples

```
<script>
x=3
if(x<0)
{
    alert ("negatif")
}
else
{
    alert ("positif")
}
</script>
```

# Conditional Statements Examples - 2

```
<script>
c=confirm("Kitap Okuyor musunuz?")
if(c)
{
    alert ("tebrikler walla")
}
else
{
    alert ("ayıp ettiniz ama")
}
</script>
```

# Conditional Statements Examples - 3

```
<script>  
p=prompt("Ankara'nın plaka numarası nedir?", " ")  
if(p=="06")  
{  
    alert("Doğru")  
}  
else  
{  
    alert("Yanlış")  
}  
</script>
```

# Built-In Functions

- **eval (expr)**
  - evaluates an expression or statement
    - eval("3 + 4"); // Returns 7 (Number)
    - eval("alert('Hello')"); // Calls the function alert('Hello')
- **isFinite (x)**
  - Determines if a number is finite
- **isNaN (x)**
  - Determines whether a value is “Not a Number”

# Built-In Functions

- **parseInt(s)**
- **parseInt(s, radix)**
  - Converts string literals to integers
  - Parses up to any character that is not part of a valid integer
    - `parseInt("3 chances")` // returns 3
    - `parseInt(" 5 alive")` // returns 5
    - `parseInt("How are you")` // returns NaN
    - `parseInt("17", 8)` // returns 15
- **parseFloat(s)**
  - Finds a floating-point value at the beginning of a string.
    - `parseFloat("3e-1 xyz")` // returns 0.3
    - `parseFloat("13.5 abc")` // returns 13.5

# Events

- An event occurs as a result of some activity
  - e.g.:
    - A user clicks on a link in a page
    - Page finished loaded
    - Mouse cursor enter an area
    - A preset amount of time elapses
    - A form is being submitted

# Event Handlers

- Event Handler – a segment of codes (usually a function) to be executed when an event occurs
- We can specify event handlers as attributes in the HTML tags.
- The attribute names typically take the form "**onXXX**" where **XXX** is the event name.
  - e.g.:  
`<a href="..." onClick="alert('Bye')">Other Website</a>`

# Event Handlers

Event Handlers	Triggered when
onChange	The value of the text field, textarea, or a drop down list is modified
onClick	A link, an image or a form element is clicked once
onDoubleClick	The element is double-clicked
onMouseDown	The user presses the mouse button
onLoad	A document or an image is loaded
onSubmit	A user submits a form
onReset	The form is reset
onUnLoad	The user closes a document or a frame
onResize	A form is resized by the user

For a complete list, see [http://www.w3schools.com/html/dom\\_obj\\_event.asp](http://www.w3schools.com/html/dom_obj_event.asp)

# onClick Event Handler Example

```
<html>
<head>
<title>onClick Event Handler Example</title>
<script type="text/javascript">
function warnUser() {
    return confirm("Are you a student?");
}
</script>
</head>
<body>
<a href="ref.html" onClick="return warnUser()">
<!--
    If onClick event handler returns false, the link
    is not followed.
-->
Students access only</a>
</body>
</html>
```

# onLoad Event Handler Example

```
<html><head>
<title>onLoad and onUnload Event Handler Example</title>
</head>
<body
  onLoad="alert('Welcome to this page')"
  onUnload="alert('Thanks for visiting this page')"
>
Load and UnLoad event test.
</body>
</html>
```

# **onMouseOver & onMouseOut Event Handler**

```
<html>
<head>
<title>onMouseOver / onMouseOut Event Handler Demo</title>
</head>
<body>
<a href="http://www.cuhk.edu.hk"
  onMouseOver="window.status='CUHK Home'; return true;"
  onMouseOut="status=''"
>CUHK</a>
</body>
</html>
```

- When the mouse cursor is over the link, the browser displays the text "CUHK Home" instead of the URL.
- The "return true;" of `onMouseOver` forces browser not to display the URL.
- `window.status` and `window.defaultStatus` are disabled in Firefox.

# onSubmit Event Handler Example

```
<html><head>
<title>onSubmit Event Handler Example</title>
<script type="text/javascript">
    function validate() {
        // If everything is ok, return true
        // Otherwise return false
    }
</script>
</head>
<body>
<form action="MessageBoard" method="POST"
    onSubmit="return validate();"
>
...
</form></body></html>
```

- If onSubmit event handler returns false, data is not submitted.
- If onReset event handler returns false, form is not reset

# Build-In JavaScript Objects

Object	Description
Array	Creates new array objects
Boolean	Creates new Boolean objects
Date	Retrieves and manipulates dates and times
Error	Returns run-time error information
Function	Creates new function objects
Math	Contains methods and properties for performing mathematical calculations
Number	Contains methods and properties for manipulating numbers.
String	Contains methods and properties for manipulating text strings

- See online references for complete list of available methods in these objects:  
<http://javascript-reference.info/>

# String Object (Some useful methods)

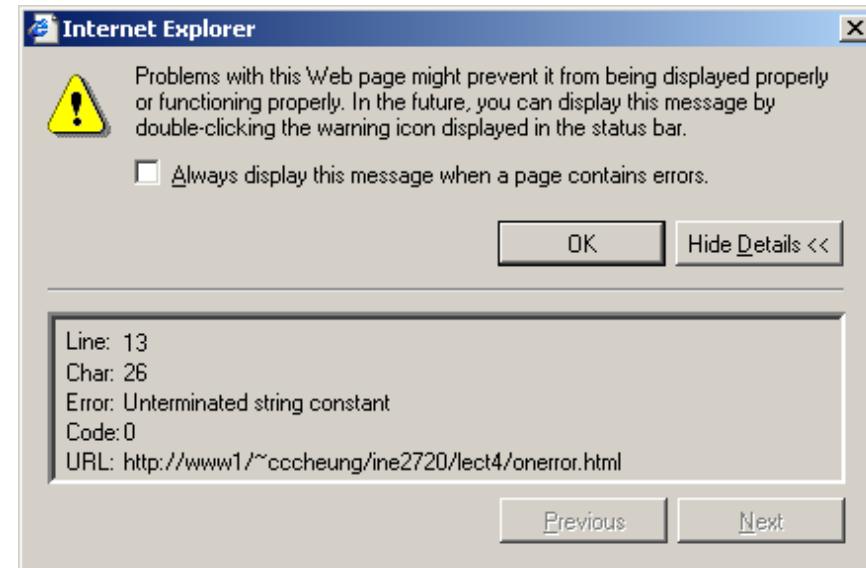
- `length`
  - A string property that tells the number of character in the string
- `charAt(idx)`
  - Returns the character at location "idx"
- `toUpperCase(), toLowerCase()`
  - Returns the same string with all uppercase/lowercase letters
- `substring(beginIdx)`
  - Returns a substring started at location "beginIdx"
- `substring(beginIdx, endIdx)`
  - Returns a substring started at "beginIdx" until "endIdx" (but not including "endIdx")
- `indexOf(str)`
  - Returns the position where "str" first occurs in the string

# Error and Exception Handling in JavaScript

- Javascript makes no distinction between Error and Exception (Unlike Java)
- Handling Exceptions
  - The `onError` event handler
    - A method associated with the window object.
    - It is called whenever an exception occurs
  - The `try ... catch ... finally` block
    - Similar to Java try ... catch ... finally block
    - For handling exceptions in a code segment
  - Use `throw` statement to throw an exception
    - You can throw value of any type
  - The `Error` object
    - Default object for representing an exception
    - Each Error object has a `name` and `message` properties

# How to use “onError” event handler?

```
<html>
<head>
<title>onerror event handler example</title>
<script type="text/javascript">
function errorHandler() {
    alert("Error Ourred!");
}
// JavaScript is casesensitive
// Don't write onerror!
window.onerror = errorHandler;
</script>
</head>
<body>
<script type="text/javascript">
    document.write("Hello there;
</script>
</body>
</html>
```



# try ... catch ... finally

```
try {  
    // Contains normal codes that might throw an exception.  
  
    // If an exception is thrown, immediately go to  
    // catch block.  
  
} catch ( errorVariable ) {  
    // Codes here get executed if an exception is thrown  
    // in the try block.  
  
    // The errorVariable is an Error object.  
  
} finally {  
    // Executed after the catch or try block finish  
  
    // Codes in finally block are always executed  
}  
// One or both of catch and finally blocks must accompany the try  
block.
```

# try ... catch ... finally example

```
<script type="text/javascript">
try{
    document.write("Try block begins<br>");
    // create a syntax error
    eval ("10 + * 5");

} catch( errVar ) {
    document.write("Exception caught<br>");
    // errVar is an Error object
    // All Error objects have a name and message properties
    document.write("Error name: " + errVar.name + "<br>");
    document.write("Error message: " + errVar.message +
                  "<br>");

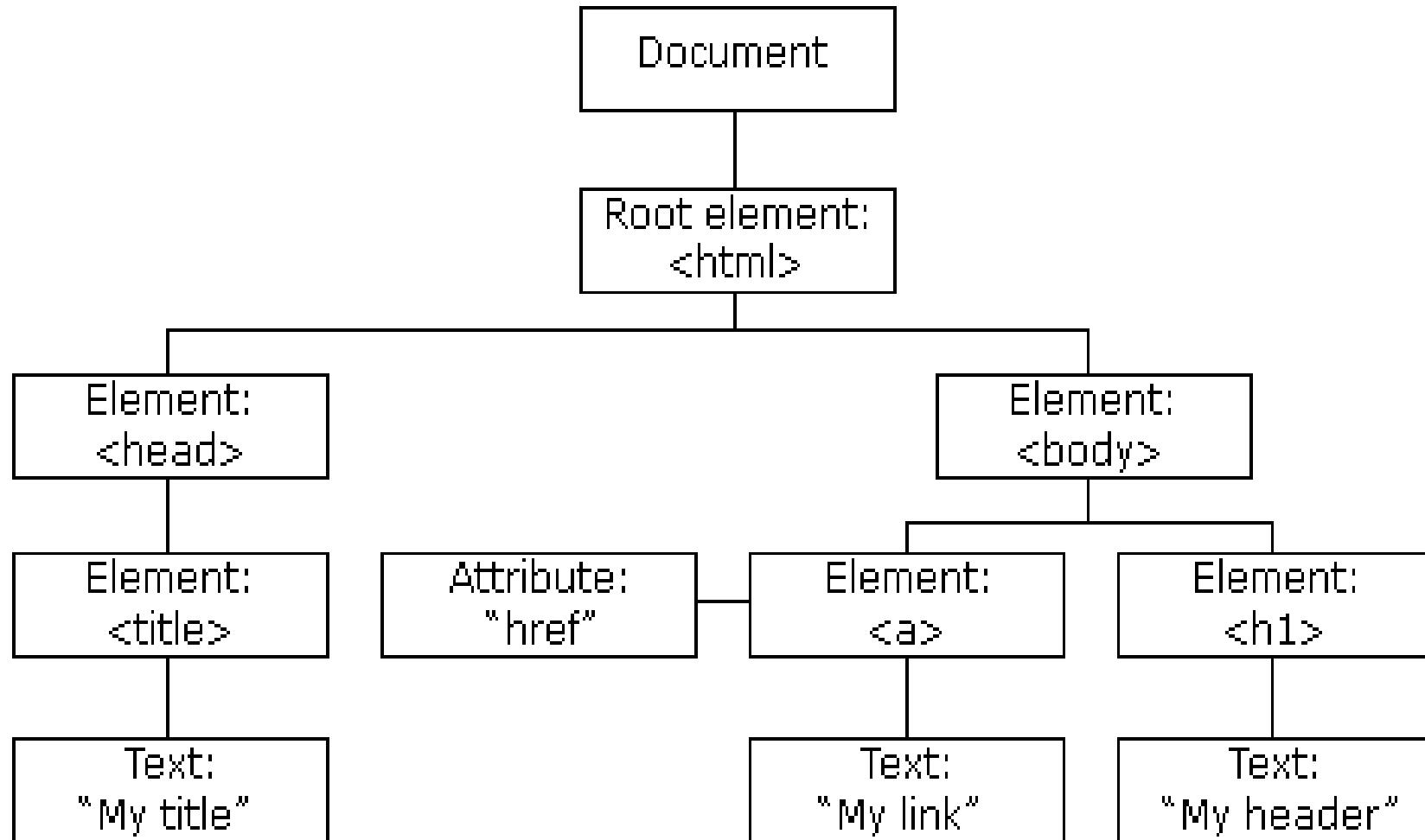
} finally {
    document.write("Finally block reached!");
}
</script>
```

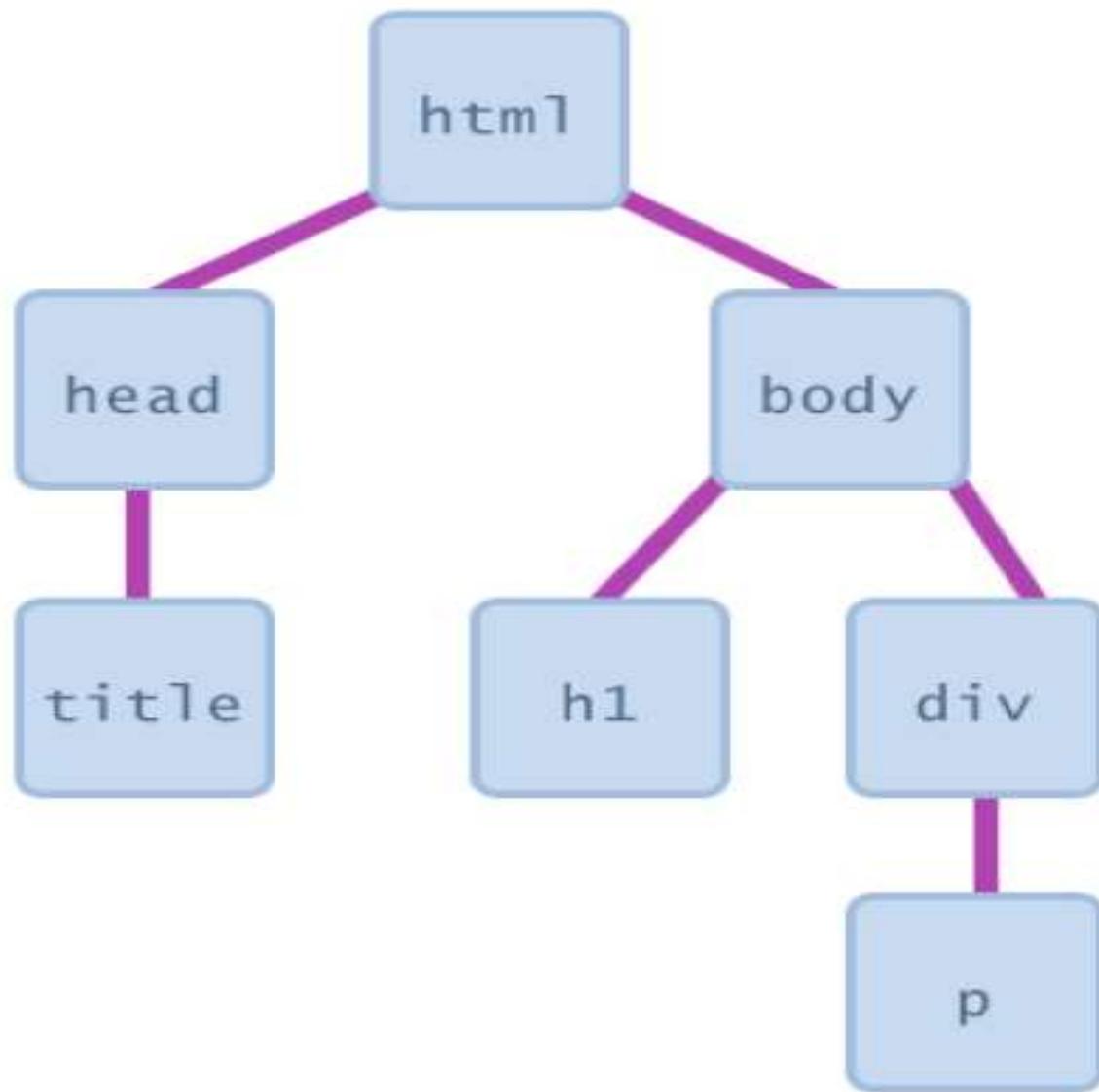
# Throwing Exception

```
<script type="text/javascript">
try{
    var num = prompt("Enter a number (1-2) : ", "1");
    // You can throw exception of any type
    if (num == "1")
        throw "Some Error Message";
    else
        if (num == "2")
            throw 123;
        else
            throw new Error ("Invalid input");
} catch( err ) {
    alert(typeof(errMsg) + "\n" + err);
    // instanceof operator checks if err is an Error object
    if (err instanceof Error)
        alert("Error Message: " + err.message);
}
</script>
```

# Document Object Model (DOM)

- When a web page is loaded, the browser creates a **Document Object Model** of the page.
- Representation of the current web page as a tree of Javascript objects
- allows you to view/modify page elements in script code after page has loaded
- client side = highly responsive interactions
- browser-independent
- allows progressive enhancement.





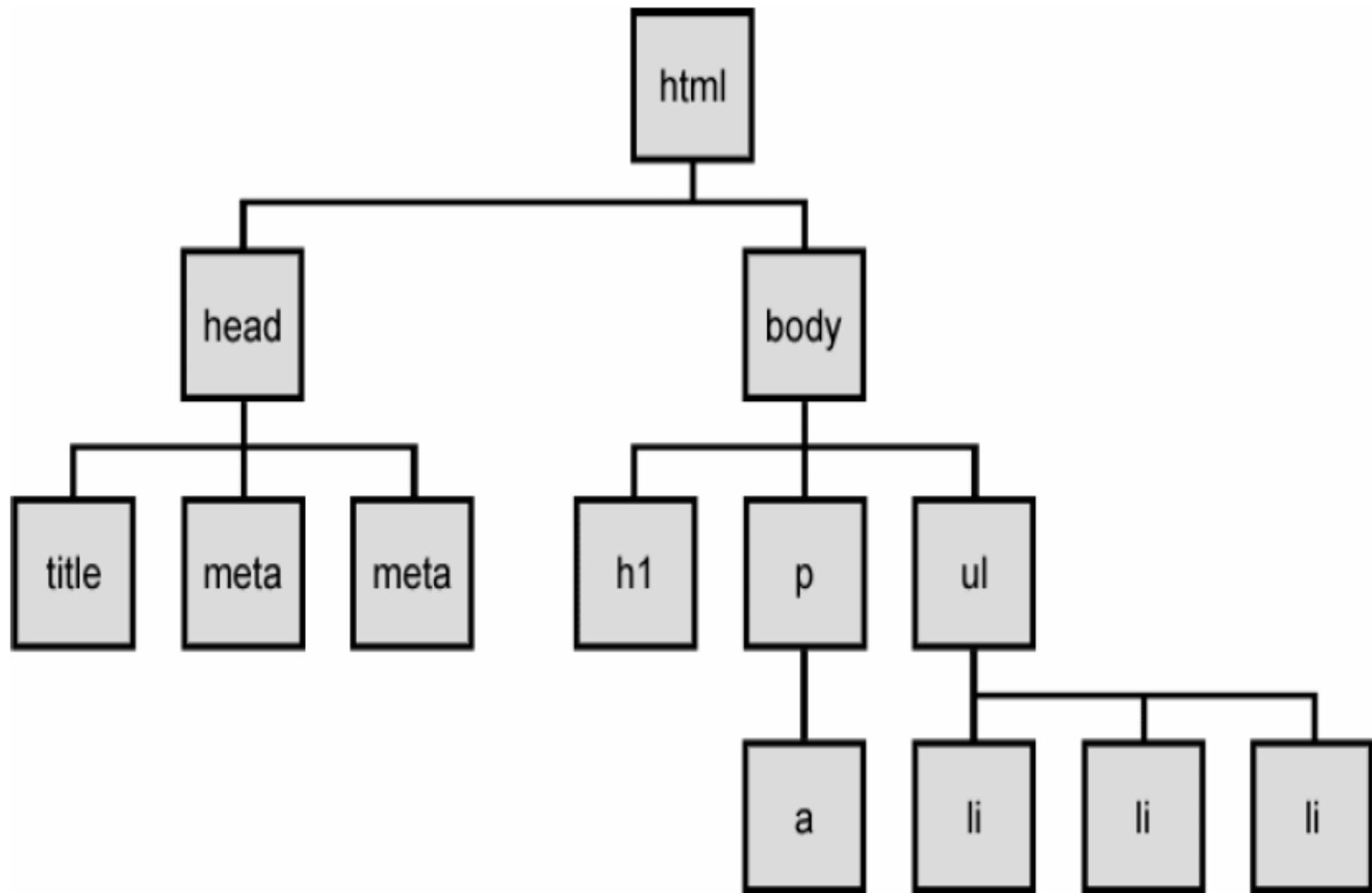
# An example XHTML page

---

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/XHTML1/DTD/XHTML1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Page Title</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<meta http-equiv="Content-Language" content="en-us" />
</head>
<body>
<h1>This is a heading</h1>
<p>A paragraph with a <a href="http://www.google.com/">link</a>.</p>
<ul>
<li>a list item</li>
<li>another list item</li>
<li>a third list item</li>
</ul>
</body>
</html>
```

# The resulting DOM tree

---



- With the object model, JavaScript gets all the power it needs to create dynamic HTML:
- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

# What is the DOM?

- The DOM is a W3C (World Wide Web Consortium) standard.
- The DOM defines a standard for accessing documents:
- *"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."*
- The W3C DOM standard is separated into 3 different parts:
- Core DOM - standard model for all document types
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

# What is the HTML DOM?

- The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:
- The HTML elements as **objects**
- The **properties** of all HTML elements
- The **methods** to access all HTML elements
- The **events** for all HTML elements

# JavaScript - HTML DOM Methods

- HTML DOM methods are **actions** you can perform (on HTML Elements).
- HTML DOM properties are **values** (of HTML Elements) that you can set or change.

# The DOM Programming Interface

- The HTML DOM can be accessed with JavaScript (and with other programming languages).
- In the DOM, all HTML elements are defined as **objects**.
- The programming interface is the properties and methods of each object.
- A **property** is a value that you can get or set (like changing the content of an HTML element).
- A **method** is an action you can do (like add or deleting an HTML element).

# Example

- The following example changes the content (the innerHTML) of the <p> element with id="demo":

```
<html>
<body>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML
= "Hello World!";
</script>
</body>
</html>
```

# The getElementById Method

- The most common way to access an HTML element is to use the id of the element.
- In the example above the getElementById method used id="demo" to find the element.

## The innerHTML Property

- The easiest way to get the content of an element is by using the **innerHTML** property.
- The innerHTML property is useful for getting or replacing the content of HTML elements.
- The innerHTML property can be used to get or change any HTML element, including <html> and <body>.

# The HTML DOM Document Object

- The document object represents your web page.
- If you want to access any element in an HTML page, you always start with accessing the document object.
- Below are some examples of how you can use the document object to access and manipulate HTML.

# JavaScript Objects

- Real Life Objects, Properties, and Methods
- In real life, a car is an **object**.
- A car has **properties** like weight and color, and **methods** like start and stop:
- **ObjectProperties**
  - 1.car.name = Fiat
  - 2.car.model = 500
  - 3.car.weight = 850kg
  - 4.car.color = white
- **Methods**
  - 1.car.start()
  - 2.car.drive()
  - 3.car.brake()
  - 4.car.stop()

- All cars have the same **properties**, but the property values differ from car to car.
- All cars have the same **methods**, but the methods are performed at different times.
- JavaScript Objects
- You have already learned that JavaScript variables are containers for data values.
- This code assigns a **simple value** (Fiat) to a **variable** named car:

- <!DOCTYPE html>
- <html>
- <body>
- <p>Creating a JavaScript Variable.</p>
- <p id="demo"></p>
- <script>
- var car = "Fiat";
- document.getElementById("demo").innerHTML = car;
- </script>
- </body>
- </html>

- Objects are variables too. But objects can contain many values.
- This code assigns **many values** (Fiat, 500, white) to a **variable** named car:
- `<!DOCTYPE html>`
- `<html>`
- `<body>`
- `<p>Creating a JavaScript Object.</p>`
- `<p id="demo"></p>`
- `<script>`
- `var car = {type:"Fiat", model:"500", color:"white"};`
- `document.getElementById("demo").innerHTML = car.type;`
- `</script>`
- `</body>`
- `</html>`
- The values are written as **name:value** pairs (name and value separated by a colon).

# Object Properties

- The name:values pairs (in JavaScript objects) are called **properties**.
- `var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};`
- **Object Methods:**
- Methods are **actions** that can be performed on objects.
- Methods are stored in properties as **function definitions**.

# Object Definition

- You define (and create) a JavaScript object with an object literal:
- Example
- **var person = {firstName:"John",  
lastName:"Doe", age:50, eyeColor:"blue"};**
- **var person = {  
    firstName:"John",  
    lastName:"Doe",  
    age:50,  
    eyeColor:"blue"  
};**

# Accessing Object Properties

- You can access object properties in two ways:
- *objectName.propertyName*

*Or*

*objectName["propertyName"]*
- Example1
  - `person.lastName;`
- Example2
  - `person["lastName"];`

- <!DOCTYPE html>
- <html>
- <body>
- <p>
- There are two different ways to access an object property:
- </p>
- <p>You can use person.property or person["property"].</p>
- <p id="demo"></p>
- <script>
- var person = {
  - firstName: "John",
  - lastName : "Doe",
  - id     : 5566
- };
- document.getElementById("demo").innerHTML =
  - person.firstName + " " + person.lastName;
- </script>
- </body>
- </html>

- <!DOCTYPE html>
- <html>
- <body>
- <p>
- There are two different ways to access an object property:
- </p>
- <p>You can use person.property or person["property"].</p>
- <p id="demo"></p>
- <script>
- var person = {
- firstName: "John",
- lastName : "Doe",
- id      : 5566
- };
- document.getElementById("demo").innerHTML =
- person["firstName"] + " " + person["lastName"];
- </script>
- </body>
- </html>

# Accessing Object Methods

- You access an object method with the following syntax:
  - *objectName.methodName()*
- Example
  - `name = person.fullName();`

- <!DOCTYPE html>
- <html>
- <body>
- <p>Creating and using an object method.</p>
- <p>An object method is a function definition, stored as a property value.</p>
- <p id="demo"></p>
- <script>
- var person = {
- firstName: "John",
- lastName : "Doe",
- id     : 5566,
- fullName : function() {
- return this.firstName + " " + this.lastName;
- }
- };
- document.getElementById("demo").innerHTML = person.fullName();
- </script>
- </body>
- </html>

# The Document Object

- A document is a web page that is being either displayed or created. The document has a number of properties that can be accessed by JavaScript programs and used to manipulate
  - the content of the page.
    - **write or writeln**
- Html pages can be created on the fly using JavaScript. This is done by using the write or writeln methods of the document object.
- Syntax:**document.write (“String”); document.writeln (“String”);**
- In this document is object name and write () or writeln () are methods. Symbol period is used as connector between object name and method name. The difference between these two methods is carriage form feed character that is new line character automatically added into the document.
- Exmaple: **document.write(“<body>”); document.write(“<h1> Hello </h1>”);**

- **bgcolor and fgcolor**

These are used to set background and foreground(text) color to webpage. The methods accept either hexadecimal values or common names for colors.

Syntax: **document.bgcolor="#1f9de1"; document\_fgcolor="silver";**

- **anchors**

The anchors property is an array of anchor names in the order in which they appear in the HTML Document. Anchors can be accessed like this:

- Syntax:

- **document.anchors[0];**

- **document.anchors[n-1];**

- **Links**

Another array holding all links in the order in which they were appeared on the Webpage

- **Forms**

Another array, this one contains all of the HTML forms. By combining this array with the individual form objects each form item can be accessed.

# The Window Object

- The window object is used to create a new window and to control the properties of window.
- Methods:
  - *open("URL","name")* : This method opens a new window which contains the document specified by URL and the new window is identified by its name.
  - *close()*: this shutdowns the current window.
- *Properties:*
- toolbar = [1|0] location= [1|0] menubar = [1|0] scrollbars = [1|0] status = [1|0] resizable = [1|0]
- where as 1 means *on* and 0 means *off*
- *height=pixels, width=pixels* : These properties can be used to set the window size.
- The following code shows how to open a new window
- *newWin =*  
*open("first.html","newWin","status=0,toolbar=0,width=100,height=100");*

- Window object supports three types of message boxes.
- Alert box      Confirm box      Prompt box
- **Alert box** is used to display warning/error messages to user. It displays a text string with *OK* button.
- Syntax: window.Alert ("Message");

- **Confirm Box** is useful when submitting form data. This displays a window containing message with two buttons: *OK* and *Cancel*. Selecting *Cancel* will abort the any pending action, while *OK* will let the action proceed.
- Syntax
- `window.confirm("String");`

- **Prompt box** used for accepting data from user through keyboard. This displays simple window that contains a prompt and a text field in which user can enter data. This method has two parameters: a text string to be used as a prompt and a string to use as the default value. If you don't want to display a default then simply use an empty string. Syntax
- Variable=window.prompt("string","default value");

# The Form Object

- Two aspects of the form can be manipulated through JavaScript. First, most commonly and probably most usefully, the data that is entered onto your form can be checked at submission. Second you can actually build forms through JavaScript.
- Form object supports three events to validate the form  
***onClick = “method()”***
  - This can be applied to all form elements. This event is triggered when the user clicks on the element.  
***onSubmit = “method()”***
    - This event can only be triggered by form itself and occurs when a form is submitted.  
***onReset = “method()”***
      - This event can only be triggered by form itself and occurs when a form is reset.

# The browser Object

- The browser is JavaScript object that can be used to know the details of browser. Some of the properties of the browser object is as follows:

Property	Description
navigator.appCodeName	It returns the internal name for the browser. For major browsers it is Mozilla
navigator.appName	It returns the public name of the browser – navigator or Internet Explorer
navigator.appVersion	It returns the version number, platform on which the browser is running.
navigator.userAgent	The strings appCodeName and appVersion concatenated together
navigator.plugins	An array containing details of all installed plug-ins
Navigator.mimeTypes	An array of all supported MIME Types

# The Math Object

- The *Math* object holds all mathematical functions and values. All the functions and attributes used in complex mathematics must be accessed via this object.
- Syntax:
- `Math.methodname();`
- `Math.value;`

Method	Description	Example
Math.abs(x)	Returns the absolute value	Math.abs(-20) is 20
Math.ceil(x)	Returns the ceil value	Math.ceil(5.8) is 6 Math.ceil(2.2) is 3
Math.floor(x)	Returns the floor value	Math.floor(5.8) is 5 Math.floor(2.2) is 2
Math.round(x)	Returns the round value, nearest integer value	Math.round(5.8) is 6 Math.round(2.2) is 2
Math.trunc(x)	Removes the decimal places it returns only integer value	Math.trunc(5.8) is 5 Math.trunc(2.2) is 2
Math.max(x,y)	Returns the maximum value	Math.max(2,3) is 3 Math.max(5,2) is 5
Math.min(x,y)	Returns the minimum value	Math.min(2,3) is 2 Math.min(5,2) is 2
Math.sqrt(x)	Returns the square root of x	Math.sqrt(4) is 2

<b>Math.pow(a,b)</b>	This method will compute the $a^b$	<b>Math.pow(2,4) is 16</b>
Math.sin(x)	Returns the sine value of x	Math.sin(0.0) is 0.0
Math.cos(x)	Returns cosine value of x	Math.cos(0.0) is 1.0
Math.tan(x)	Returns tangent value of x	Math.tan(0.0) is 0
Math.exp(x)	Returns exponential value i.e $e^x$	Math.exp(0) is 1
Math.random(x)	Generates a random number in between 0 and 1	Math.random()
Math.log(x)	Display logarithmic value	Math.log(2.7) is 1
Math.PI	Returns a $\pi$ value	a = Math.PI; a = 3.141592653589793

# The Date Object

- This object is used for obtaining the date and time. In JavaScript, dates and times represent in milliseconds since 1<sup>st</sup> January 1970 UTC. JavaScript supports two time zones: UTC and local. UTC is Universal Time, also known as Greenwich Mean Time(GMT), which is standard time throughout the world. Local time is the time on your System. A JavaScript *Date* represents date from -1,000,000,000 to - 1,000,000,000 days relative to 01/01/1970.
- Date Object Constructors:
- *new Date();* Constructs an empty date object.
- *new Date("String");* Creates a Date object based upon the contents of a text string.
- *new Date(year, month, day[,hour, minute, second] );* Creates a Date object based upon the numerical values for the year, month and day.
- `var dt=new Date();`
- `document.write(dt);`      **// Tue Dec 23 11:23:45 UTC+0530 2015**

- <!DOCTYPE html>
- <html>
- <body>
- <p id="demo"></p>
- <script>
- document.getElementById("demo").innerHTML =  
Date();
- </script>
- </body>
- </html>

output:

Mon Jan 29 2018 09:20:16 GMT+0530 (India  
Standard Time)

# JavaScript Date Methods

- Date methods let you get and set date values (years, months, days, hours, minutes, seconds, milliseconds)

Method	Description
getDate()	Get the day as a number (1-31)
getDay()	Get the weekday as a number (0-6)
getFullYear()	Get the four digit year (yyyy)
getHours()	Get the hour (0-23)
getMilliseconds()	Get the milliseconds (0-999)
getMinutes()	Get the minutes (0-59)
getMonth()	Get the month (0-11)

# JavaScript Arrays

JavaScript arrays are used to store multiple values in a single variable.

## Displaying Arrays

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars[0];
</script>

</body>
</html>
```

## Creating an Array

Using an array literal is the easiest way to create a JavaScript Array.

### Syntax:

```
var array-name = [item1, item2, ...];
```

## Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

```
var cars = new Array("Saab", "Volvo", "BMW");
```

## Access the Elements of an Array

You refer to an array element by referring to the **index number**.

```
var name = cars[0];
```

This statement modifies the first element in cars:

```
cars[0] = "Opel";
```

## Can Have Different Objects in One Array

- JavaScript variables can be objects. Arrays are special kinds of objects.
- Because of this, you can have variables of different types in the same Array.

### Example:

```
myArray[0] = Date.now;  
myArray[1] = myFunction;  
myArray[2] = myCars;
```

# Arrays are Objects

- Arrays are a special type of objects. The **typeof** operator in JavaScript returns "object" for arrays.
  - But, JavaScript arrays are best described as arrays.

Arrays use **numbers** to access its "elements". In this example, person[0] returns John:

Array: var person = ["John", "Doe", 46];

Objects use **names** to access its "members". In this example, `person.firstName` returns `John`:

```
Object: var person = {firstName:"John", lastName:"Doe",  
                    age:46};
```

# Array Properties and Methods

The real strength of JavaScript arrays are the built-in array properties and methods:

## Examples

```
var x = cars.length; // The length property returns the number  
                     of elements in cars
```

```
var y = cars.sort();    // The sort() method sort cars in  
                        alphabetical order
```

**Note:**

- ❖ var points = new Array(); // Bad
- ❖ var points = [];// Good

**When to Use Arrays? When to use Objects?**

- JavaScript does not support associative arrays.
- You should use objects when you want the element names to be strings.
- You should use arrays when you want the element names to be sequential numbers.

# Array Object Methods

Cont..

Method	Description
<a href="#">concat()</a>	Joins two or more arrays, and returns a copy of the joined arrays
<a href="#">indexOf()</a>	Search the array for an element and returns its position
<a href="#">join()</a>	Joins all elements of an array into a string
<a href="#">lastIndexOf()</a>	Search the array for an element, starting at the end, and returns its position
<a href="#">pop()</a>	Removes the last element of an array, and returns that element
<a href="#">push()</a>	Adds new elements to the end of an array, and returns the new length
<a href="#">reverse()</a>	Reverses the order of the elements in an array
<a href="#">shift()</a>	Removes the first element of an array, and returns that element
<a href="#">slice()</a>	Selects a part of an array, and returns the new array
<a href="#">sort()</a>	Sorts the elements of an array
<a href="#">splice()</a>	Adds/Removes elements from an array
<a href="#">toString()</a>	Converts an array to a string, and returns the result
<a href="#">unshift()</a>	Adds new elements to the beginning of an array, and returns the new length
<a href="#">valueOf()</a>	Returns the primitive value of an array

**Concat():** Joins two or more arrays, and returns a copy of the joined arrays.

**Syntax:** `array.concat(array1,array2.....);`

**Ex:**

```
var hege = ["Cecilie", "Lone"];
var stale = ["Emil", "Tobias", "Linus"];
var kai = ["Robin"];
var children = hege.concat(stale,kai);
```

**indexOf():** this method searches the array for the specified item, and returns its position.

- The search will start at the specified position, or at the beginning if no start position is specified, and end the search at the end of the array.

Returns -1 if the item is not found.

If the item is present more than once, the indexOf method returns the position of the first occurrence.

**Syntax:** `array.indexOf(item);`

**Ex:**

```
var fruits=["Banana","Orange","Apple","Mango"];
var x=fruits.indexOf("Apple");
```

Here the value of x is:2 i.e. Apple is at position 2.

**Join():** The join() method joins the elements of an array into a string, and returns the string.

The elements will be separated by a specified separator. The default separator is comma (,).

**Syntax:** array.join();

**Ex:** var fruits=["Banana" Orange","Apple","Mango"];  
var x=fruits.join();

the result of x will be Banana,Orange,Apple,Mango.

**Slice():** The slice() method returns the selected elements in an array, as a new array object.

The slice() method selects the elements starting at the given *start* argument, and ends at, *but does not include*, the given *end* argument.

**Syntax:** array.slice (start, end);

**Ex:** var fruits=["Banana",Orange,"Apple","Mango"];  
var x=fruits.slice(1,3);

the result of x will be [Orange,Apple];

**Splice():** The splice() method adds/removes items to/from an array, and returns the removed item(s).

**Syntax:** array.splice(index,howmany,item1,item2....);

**Ex:** var fruits=["Banana",Orange,"Apple","Mango"];  
fruits.splice(2,1,"Lemon","Kiwi");

the result of fruits will be [Banana,Orange,Lemon,Kiwi,Mango]

## Arrays Operations and Properties

- Declaring new empty array:

```
var arr = new Array();
```

- Declaring an array holding few elements:

```
var arr = [1, 2, 3, 4, 5];
```

- Appending an element / getting the last element:

```
arr.push(3);
```

```
var element = arr.pop();
```

- Reading the number of elements (array length):

```
arr.length;
```

- Finding element's index in the array:

```
arr.indexOf(1);
```

# JavaScript Regular Expressions

- A regular expression is a sequence of characters that forms a search pattern.
- The search pattern can be used for text search and text replace operations.

## Syntax

*/pattern/modifiers;*

## Example:

```
var patt = /w3schools/I
```

- ✓ **/w3schools/i** is a regular expression.
- ✓ **w3schools** is a pattern (to be used in a search).
- ✓ **i** is a modifier (modifies the search to be case-insensitive)

## Using String Methods

In JavaScript, regular expressions are often used with the two **string methods**: search() and replace().

- ❖ **The search(pattern) method** uses an expression to search for a match, and returns the position of the match.
- ❖ **The replace(pattern1, pattern2) method** returns a modified string where the pattern is replaced.
- ❖ **The split(pattern) method**
- ❖ **The match(pattern) method** searches for a matching pattern.  
Returns an array holding the results, or null if no match is found.

### Example

```
var str = "Visit W3Schools";  
var n = str.search(/w3schools/i);
```

The result in n will be: 6

### Example

```
var str = "Visit Microsoft!";  
var res = str.replace(/microsoft/i, "W3Schools");
```

The result in res will be: Visit W3Schools!

# Regular Expression Modifiers

**Modifiers** can be used to perform case-insensitive more global searches:

<b>Modifier</b>	<b>Description</b>
i	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
m	Perform multiline matching

# Regular Expression Patterns

**Brackets** are used to find a range of characters:

<b>Expression</b>	<b>Description</b>
[abc]	Find any of the characters between the brackets
[0-9]	Find any of the digits between the brackets
(x y)	Find any of the alternatives separated with

**Metacharacters** are characters with a special meaning:

**Cont..**

Metacharacter / Token	Description
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning or at the end of a word
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx
^	Match at the start of the input string
\$	Match at the end of the input string
*	Match 0 or more times
+	Match 0 or more times
?	Match 0 or 1 times
{n}	Match the string n times
\D	Match anything except for digits

Metacharacter / Token	Description
\w	Match any alphanumeric character or the underscore
\W	Match anything except alphanumeric characters or underscore
\S	Match anything except for whitespace characters
[...]	Creates a set of characters, one of which must match if the operation is to be successful. If you need a range of characters then separate the first and last with a hyphen: [0-9] or [D-G]
[^...]	Creates a set of characters which doesnot match. If any character in the set matches then the operation has failed. This fails if any lowercase letter from d to q is matched: [^d-q].

## Quantifiers define quantities:

Quantifier	Description	
n+	Matches any string that contains at least one n	
n*	Matches any string that contains zero or more occurrences of n	
n?	Matches any string that contains zero or one occurrences of n	Cont..

## Using the RegExp Object

- In JavaScript, the RegExp object is a regular expression object with predefined properties and methods.

### Using test() / using exec()

- The test() method is a RegExp expression method.
- It searches a string for a pattern, and returns true or false, depending on the result.

The following example searches a string for the character "e":

### Example

```
var patt = /e/;  
        patt.test("The best things in life are free!");  
                (or)  
/e/.test("The best things in life are free!")
```

Since there is an "e" in the string, the output of the code above will be: true

# **constructor**

- It returns a reference to the array function that created the instance's prototype.
- **Syntax**
- Its syntax is as follows:

RegExp.constructor

- **Return Value**
- Returns the function that created this object's instance.

- **Example**

Try the following example program.

```
<html>
<head>
<title>JavaScript RegExp constructor Property</title>
</head>
<body>
<script type="text/javascript">
var re = new RegExp( "string" );
document.write("re.constructor is:" + re.constructor);
</script>
</body>
</html>
```

### **Output**

- re.constructor is:function RegExp() { [native code]

# **global**

**global** is a read-only boolean property of RegExp objects. It specifies whether a particular regular expression performs global matching, i.e., whether it was created with the "g" attribute.

## **Syntax**

Its syntax is as follows:

**RegExpObject.global**

## **Return Value**

Returns "TRUE" if the "g" modifier is set, "FALSE" otherwise.

```
<html>
<head>
<title>JavaScript RegExp global Property</title>
</head>
<body>
<script type="text/javascript">
var re = new RegExp( "string" );
if ( re.global ){
document.write("Test1 - Global property is set");
}else{
document.write("Test1 - Global property is not set");
}
re = new RegExp( "string", "g" );
if ( re.global ){
document.write("<br />Test2 - Global property is set");
}else{
document.write("<br />Test2 - Global property is not set");
}
</script>
</body>
</html>
```

## Output

**Test1 - Global property is not set**  
**Test2 - Global property is set**

# **ignoreCase**

**ignoreCase** is a **read-only boolean property of RegExp objects**. It specifies whether a particular regular expression performs case-insensitive matching, i.e., whether it was created with the "i" attribute.

- **Syntax**

Its syntax is as follows:

- **RegExpObject.ignoreCase**

- **Return Value**

Returns "TRUE" if the "i" modifier is set, "FALSE" otherwise.

```
<html>
<head>
<title>JavaScript RegExp ignoreCase Property</title>
</head>
<body>
<script type="text/javascript">
var re = new RegExp( "string" );
if ( re.ignoreCase ){
document.write("Test1 - ignoreCase property is set");
}else{
document.write("Test1 - ignoreCase property is not set");
}
re = new RegExp( "string", "i" );
if ( re.ignoreCase ){
document.write("<br />Test2 - ignoreCase property is set");
}else{
document.write("<br />Test2 - ignoreCase property is not set");
}
</script>
</body>
</html>
```

# **Output**

Test1 - ignoreCase property is not set

Test2 - ignoreCase property is set

# **lastIndex**

**lastIndex is a read/write property of RegExp objects. For regular expressions with the "g" attribute set, it contains an integer that specifies the character position immediately following the last match found by the `RegExp.exec()` and `RegExp.test()` methods. These methods use this property as the starting point for the next search they conduct.**

This property is read/write, so you can set it at any time to specify where in the target string, the next search should begin. **exec()** and **test()** automatically reset the **lastIndex** to 0 when they fail to find a match (or another match).

## Syntax

Its syntax is as follows:

RegExpObject.lastIndex

## Return Value

Returns an integer that specifies the character position immediately following the last match.

```
<html>
<head>
<title>JavaScript RegExp lastIndex Property</title>
</head>
<body>
<script type="text/javascript">
var str = "Javascript is an interesting scripting language";
var re = new RegExp( "script", "g" );
re.test(str);
document.write("Test 1 - Current Index: " + re.lastIndex);
re.test(str);
document.write("<br />Test 2 - Current Index: " + re.lastIndex);
</script>
</body>
</html>
```

## Output

**Test 1 - Current Index: 10**  
**Test 2 - Current Index: 35**

# **multiline**

**multiline** is a read-only boolean property of **RegExp** objects. It specifies whether a particular regular expression performs multiline matching, i.e., whether it was created with the "m" attribute.

## **Syntax**

Its syntax is as follows:

`RegExpObject.multiline`

## **Return Value**

Returns "TRUE" if the "m" modifier is set, "FALSE" otherwise.

```
<html>
<head>
<title>JavaScript RegExp multiline Property</title>
</head>
<body>
<script type="text/javascript">
var re = new RegExp( "string" );
if ( re.multiline ){
document.write("Test1-multiline property is set");
}else{
document.write("Test1-multiline property is not set");
}
re = new RegExp( "string", "m" );
if ( re.multiline ){
```

```
document.write("<br/>Test2-multiline property is set");
}else{
document.write("<br/>Test2-multiline property is not
    set");
}
</script>
</body>
</html>
```

## **Output**

Test1-multiline property is not set  
Test2-multiline property is set

# **source**

**source** is a read-only string property of RegExp objects. It contains the text of the RegExp pattern. This text does not include the delimiting slashes used in regular-expression literals, and it does not include the "g", "i", and "m" attributes.

## **Syntax**

Its syntax is as follows:

RegExpObject.source

## **Return Value**

Returns the text used for pattern matching.

```
<html>
<head>
<title>JavaScript RegExp source Property</title>
</head>
<body>
<script type="text/javascript">
var str = "Javascript is an interesting scripting language";
var re = new RegExp( "script", "g" );
re.test(str);
document.write("The regular expression is : " + re.source);
</script>
</body>
</html>
```

## Output

The regular expression is : script

# Exception Handling

## Try... Catch and Throw statements

### Catching errors in JavaScript:

It is very important that the errors thrown must be caught or trapped so that they can be handled more efficiently and conveniently and the users can move better through the web page.

### Using try...catch statement:

The try..catch statement has two blocks in it:

- try block
- catch block

In the try block, the code contains a block of code that is to be tested for errors.

The catch block contains the code that is to be executed if an error occurs. The general syntax of try..catch statement is as follows:

# Exception Handling

throw

Example:

    do something

    if an error happens

{

        create a new exception object

        throw the exception

}

try.....

Example: try

{

    statement one

    statement two

    statement three

}

catch

Example:

    catch exception

{

        handle the exception }

# The concept of try...catch statement

```
<html>
<head>
<script type="text/javascript">
try
{
document.write(junkVariable)
}
catch(err)
{
document.write(err.message + "<br/>")
}
</script>
</head>
<body>
</body>
</html>
```

- The output of the above program is ‘junkVariable’ is undefined
- In the above program, the variable *junkVariable* is undefined and the usage of this in try block gives an error. The control is transferred to the catch block with this error and this error message is printed in the catch block.

# **throw in JavaScript:**

There is another statement called throw available in JavaScript that can be used along with try...catch statements to throw exceptions and thereby helps in generating. General

- **syntax of this throw statement is as follows:**  
**throw(exception)**
- *exception can be any variable of type integer or boolean or string.*

```
<html>
<head>
<script type="text/javascript">
try
{
varexfor=10
if(exfor!=20)
{
throw "PlaceError"
}
}
catch(err)
{
if(err == "PlaceError")
document.write ("Example to illustrate Throw
Statement: Variable exfor not equal to 20.
<br/>")
}
</script>
</head>
<body>
</body>
</html>
```

The output of the above program is:

Example to illustrate Throw Statement: Variable exfor not equal to 20.

In the above example program, the try block has the variable exfor initialized to 10. Using the if statement, the variable value is checked to see whether it is equal to 20. Since exfor is not equal to 20, the exception is thrown using the throw statement. This is named Place Error and the control transfers to the catch block. The error caught is checked and since this is equal to the PlaceError, the statement placed inside the error message is displayed and the output is displayed as above.

**Example:** <!DOCTYPE html>

```
<html>
<body>
<p>Please input a number between 5 and 10:</p>
<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="message"></p>
<script>
function myFunction() {
    var message, x;
    message = document.getElementById("message");
    message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
        if(x == "") throw "is Empty";
        if(isNaN(x)) throw "not a number";
        if(x > 10) throw "too high";
        if(x < 5) throw "too low";
    }
    catch(err) {
        message.innerHTML = "Input " + err;
    }
}
</script></body></html>
```

Please input a number between 5 and 10:

Input too high

## Form validation program example

```
<html>
<head>
<script language="javascript">
function validate()
{
    var x=f1.t1.value;
    var y=f1.t2.value;
    var re=/\d|\W|_/g;

    if(x==" " || y==" ")
        window.alert("Enter both user id and password");
    else if(x.length<6 || y.length<6)
        window.alert("both user id and password should greater than or equal to 6");
    else if((re.test(x)))
        window.alert("user name should be alphabets");
    else
        window.alert("welcome"+ " Mr." +x);
}
```

```
</script>
</head>
<body bgcolor="green">
<form name="f1">
<center>
User Name<input type="text" name="t1"><br><br>
Password <input type="password" name="t2"><br><br>
<input type="submit" value="submit" onClick=validate()>
<input type="reset" value="reset">
</center>
</form>
</body>

</html>
```

# DYNAMIC HTML

- **DHTML** is combination of HTML, CSS and JavaScript. It gives pleasant appearance to web page.
- Difference between HTML and DHTML

HTML	DHTML
HTML is used to create static web pages.	DHTML is used to create dynamic web pages.
HTML consists of simple html tags.	DHTML is made up of HTML tags+cascading style sheets+javascript.
Creation of html web pages is simplest but less interactive.	Creation of DHTML is complex but more interactive.

**Dynamic HTML, or DHTML, is for a collection of technologies used together to create interactive and animated by using a combination of a static markup language (such as HTML), a client-side scripting language (such as JavaScript), a presentation definition language (such as CSS), and the Document Object Model.**

- **Uses**
- DHTML allows authors to add effects to their pages that are otherwise difficult to achieve.
- For example, DHTML allows the page author to:
- Animate text and images in their document, independently moving each element from any starting point to any ending point, following a predetermined path or one chosen by the user.
- • Embed a ticker that automatically refreshes its content with the latest news, stock quotes, or other data.
- • Use a form to capture user input, and then process and respond to that data without having to send data back to the server.
- • Include rollover buttons or drop-down menus.

- **DHTML FORMS**
- Forms are key components of all Web-based applications. But important as they are, Web developers often present users with forms that are difficult to use.
- There are three common problems:
- Forms can be too long. A seemingly endless list of questions is to make sure the user click the back button or jump to another site.
- In many situations a specific user will need to fill out only some of the form elements. If you present needless questions to a user, you'll add clutter to your page and encourage the user to go elsewhere.
- Form entries often need to conform to certain formats and instructions. Adding this information to a Web page can make for a cluttered and unappealing screen.

## Summary of the unit:

- 1.DOM
- 2.Objects in java script
- 3.Arrays concept
- 4.RegEx
- 5.Exception Handling-try,catch and throw
- 6.DHTML-HTML+CSS+JAVASCRIPT

# Servlet

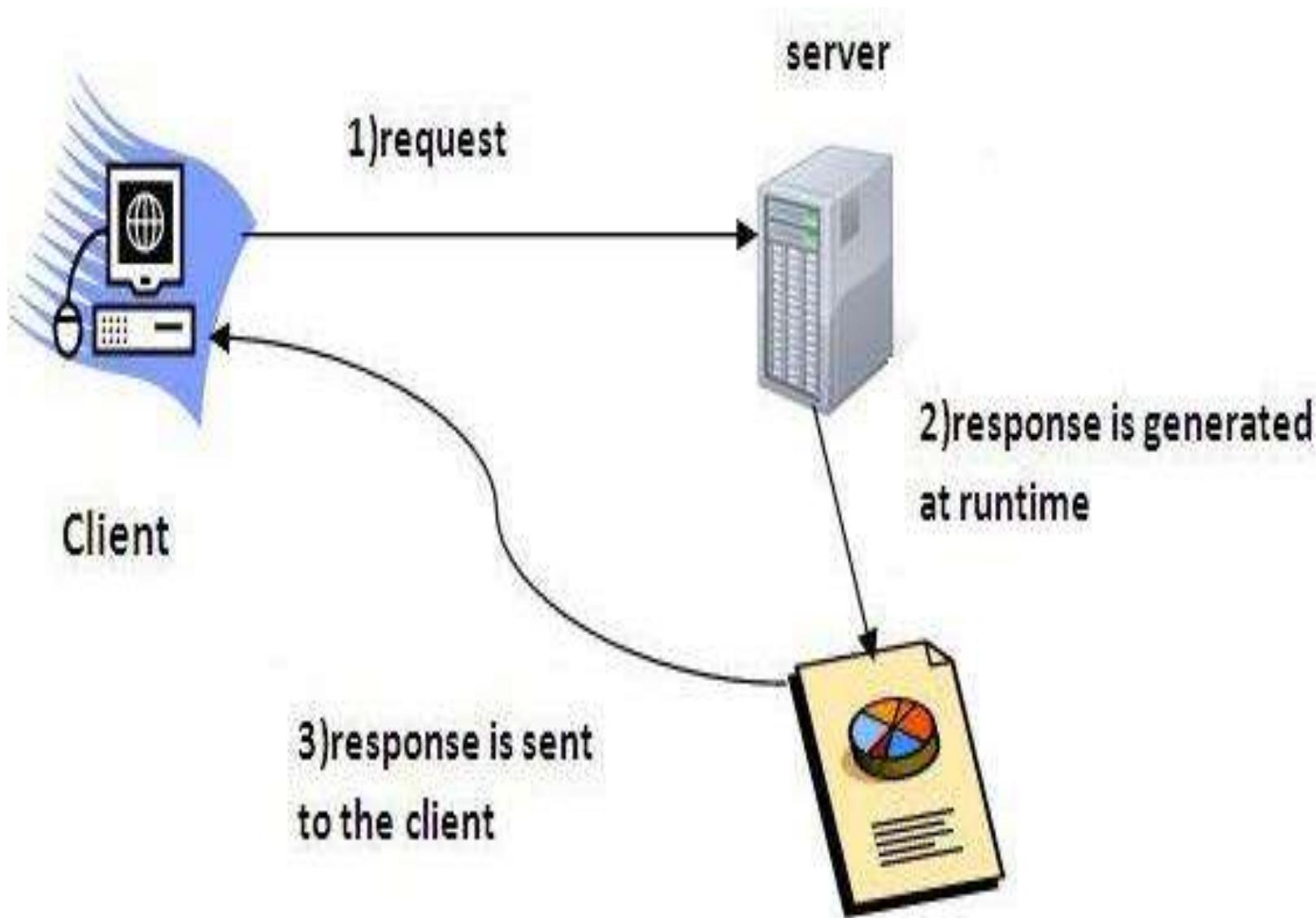
**Servlet** technology is used to create web application (resides at server side and generates dynamic web page).

- Servlet is a java based server side web technology allows us to develop dynamic web component having ability to develop dynamic web pages.
- Servlet is a JEE module based technology executes at server side having capability to extend functionality to web server,application server.
- Servlet is a JEE module web technology(server side) that allows to develop single instance multiple thread based server based technology web component.
- Servlet is JEE module web technology given set of rules and guidelines for vendor companies to develop servlet container & given application for programmers to develop server side web components.

- **Servlet** technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was popular as a server-side programming language. But there were many disadvantages of this technology. We have discussed these disadvantages below.
- There are many interfaces and classes in the servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse etc.

# What is a Servlet?

1. Servlet can be described in many ways, depending on the context.
2. Servlet is a technology i.e. used to create web application.
3. Servlet is an API that provides many interfaces and classes including documentations.
4. Servlet is an interface that must be implemented for creating any servlet.
5. Servlet is a class that extend the capabilities of the servers and respond to the incoming request. It can respond to any type of requests.
6. Servlet is a web component that is deployed on the server to create dynamic web page.

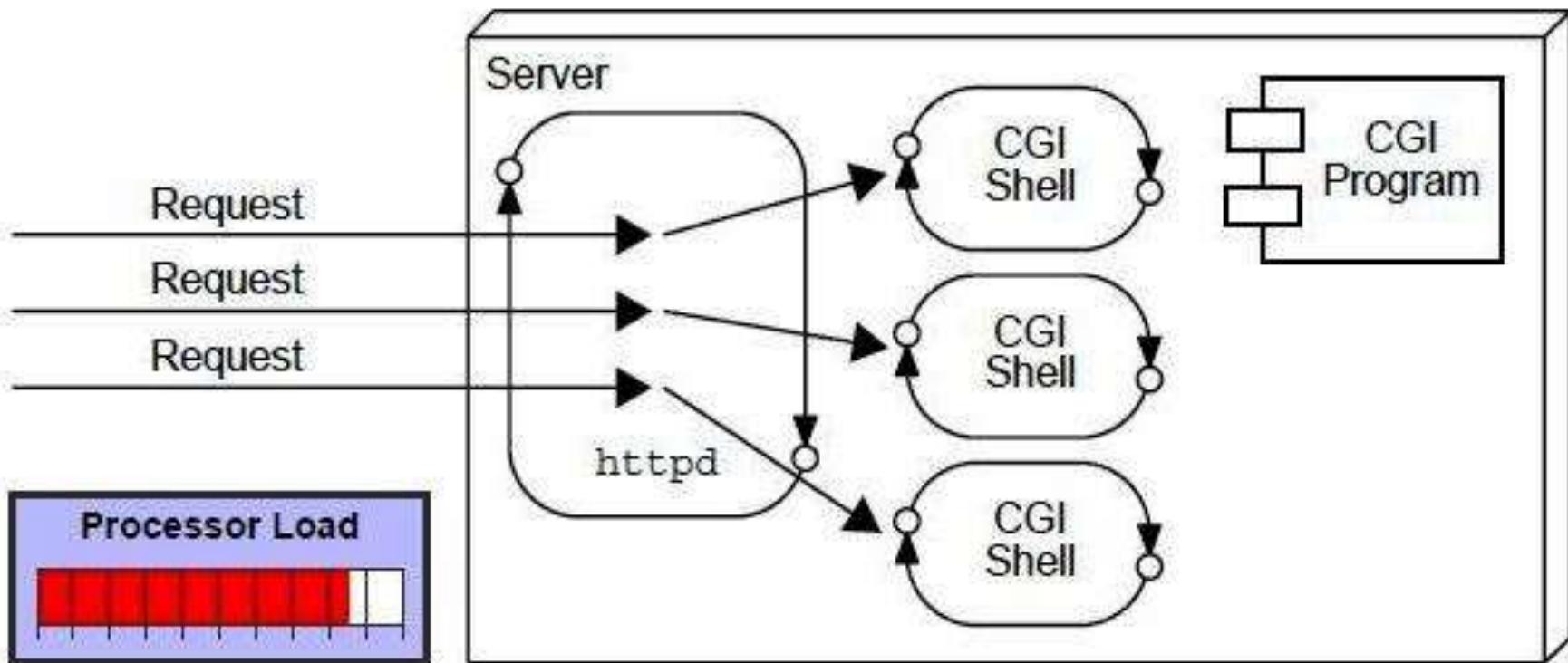


# What is web application?

- A web application is an application accessible from the web.
- A web application is composed of web components like Servlet, JSP, Filter etc. and other components such as HTML.
- The web components typically execute in Web Server and respond to HTTP request.

# CGI(Common Gateway Interface)

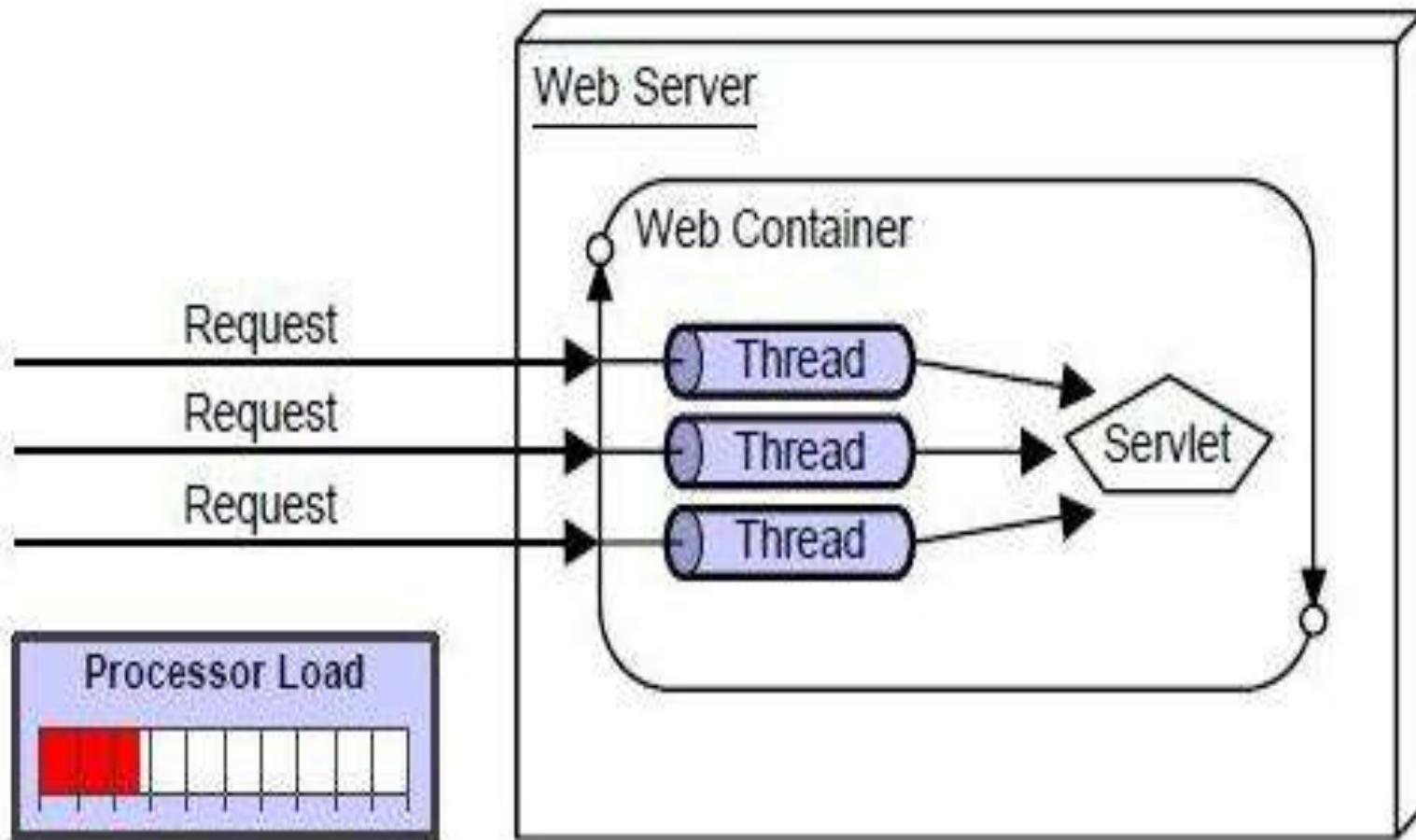
CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.



# Disadvantages of CGI

- There are many problems in CGI technology:
- If number of clients increases, it takes more time for sending response.
- For each request, it starts a process and Web server is limited to start processes.
- It uses platform dependent language e.g. C, C++, perl.

# Advantage of Servlet



- There are many advantages of servlet over CGI. The web container creates threads for handling the multiple requests to the servlet.
- Threads have a lot of benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The basic benefits of servlet are as follows:
  - **better performance:** because it creates a thread for each request not process.
  - **Portability:** because it uses java language.
  - **Robust:** Servlets are managed by JVM so we don't need to worry about memory leak, garbage collection etc.
  - **Secure:** because it uses java language..

# Web Terminology

Servlet Terminology	Description
<a href="#"><u>Website: static vs dynamic</u></a>	It is a collection of related web pages that may contain text, images, audio and video.
<a href="#"><u>HTTP</u></a>	It is the data communication protocol used to establish communication between client and server.
<a href="#"><u>HTTP Requests</u></a>	It is the request send by the computer to a web server that contains all sorts of potentially interesting information.
<a href="#"><u>Get vs Post</u></a>	It give the difference between GET and POST request.
<a href="#"><u>Container</u></a>	It is used in java for dynamically generate the web pages on the server side.
<a href="#"><u>Server: Web vs Application</u></a>	It is used to manage the network resources and for running the program or software that provides services.
<a href="#"><u>Content Type</u></a>	It is HTTP header that provides the description about what are you sending to the browser.

# Website

- Website is a collection of related web pages that may contain text, images, audio and video.
- The first page of a website is called home page. Each website has specific **internet address (URL)** that you need to enter in your browser to access a website.
- Website is hosted on one or more servers and can be accessed by visiting its homepage using a computer network.
- A website is managed by its owner that can be an **individual, company or an organization**.

# A website can be of two types:

## 1. Static Website

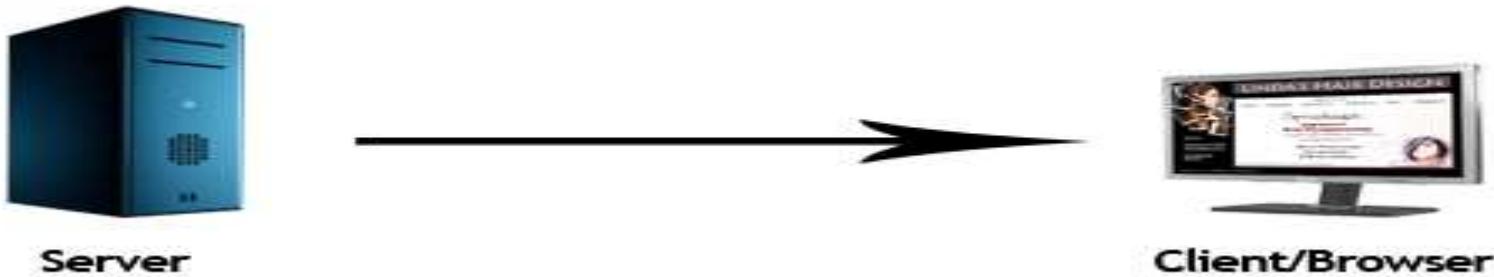
Static website is the basic type of website that is easy to create.

You don't need web programming and database design to create a static website. Its web pages are coded in HTML.

## 2. Dynamic Website

- Dynamic website is a collection of **dynamic web pages whose content changes dynamically**. It accesses content from a database or Content Management System (CMS). Therefore, when you alter or update the content of the database, the content of the website is also altered or updated.
- Dynamic website uses client-side scripting or server-side scripting, or both to generate dynamic content.

## **Static Website**



## **Dynamic Website**



Static Website	Dynamic Website
Prebuilt content is same every time the page is loaded.	Content is generated quickly and changes regularly.
It uses the <b>HTML</b> code for developing a website.	It uses the server side languages such as <b>PHP, SERVLET, JSP, and ASP.NET</b> etc. for developing a website.
It sends exactly the same response for every request.	It may generate different HTML for each of the request.
The content is only changes when someone publishes and updates the file (sends it to the web server).	The page contains "server-side" code it allows the server to generate the unique content when the page is loaded.
Flexibility is the main advantage of static website.	Content Management System (CMS) is the main advantage of dynamic website.

# HTTP (Hyper Text Transfer Protocol)

- The Hypertext Transfer Protocol (HTTP) is application-level protocol for collaborative, distributed, hypermedia information systems.
- It is the data communication protocol used to establish communication between client and server.
- HTTP is TCP/IP based communication protocol, which is used to deliver the data like image files, query results, HTML files etc on the World Wide Web (WWW) with the default port is TCP 80.
- It provides the standardized way for computers to communicate with each other.

- **The Basic Characteristics of HTTP (Hyper Text Transfer Protocol):**
- It is the protocol that allows web servers and browsers to exchange data over the web.
- It is a request response protocol.
- It uses the reliable TCP connections by default on TCP port 80.
- It is stateless means each request is considered as the new request. In other words, server doesn't recognize the user by default.

- **The Basic Features of HTTP (Hyper Text Transfer Protocol):**
- There are three fundamental features that make the HTTP a simple and powerful protocol used for communication:
  - **HTTP is media independent:** It refers to any type of media content can be sent by HTTP as long as both the server and the client can handle the data content.
  - **HTTP is connectionless:** It is a connectionless approach in which HTTP client i.e., a browser initiates the HTTP request and after the request is sends the client disconnects from server and waits for the response.
  - **HTTP is stateless:** The client and server are aware of each other during a current request only. Afterwards, both of them forget each other. Due to the stateless nature of protocol, neither the client nor the server can retain the information about different request across the web pages.

# HTTP Requests

HTTP Request	Description
<b>GET</b>	Asks to get the resource at the requested URL.
<b>POST</b>	Asks the server to accept the body info attached. It is like GET request with extra info sent with the request.
<b>HEAD</b>	Asks for only the header part of whatever a GET would return. Just like GET but with no body.
<b>TRACE</b>	Asks for the loopback of the request message, for testing or troubleshooting.
<b>PUT</b>	Says to put the enclosed info (the body) at the requested URL.
<b>DELETE</b>	Says to delete the resource at the requested URL.
<b>OPTIONS</b>	Asks for a list of the HTTP methods to which the thing at the request URL can respond

# GET and POST

- Two common methods for the request-response between a server and client are:
- **GET**- It requests the data from a specified resource
- **POST**- It submits the processed data to a specified resource

# Get vs. Post

GET	POST
1) In case of Get request, only <b>limited amount of data</b> can be sent because data is sent in header.	In case of post request, <b>large amount of data</b> can be sent because data is sent in body.
2) Get request is <b>not secured</b> because data is exposed in URL bar.	Post request is <b>secured</b> because data is not exposed in URL bar.
3) Get request <b>can be bookmarked.</b>	Post request <b>cannot be bookmarked.</b>
4) Get request is <b>idempotent</b> . It means second request will be ignored until response of first request is delivered	Post request is <b>non-idempotent</b> .
5) Get request is <b>more efficient</b> and used more than Post.	Post request is <b>less efficient</b> and used less than get.

# Anatomy of Get Request

- The query string (name/value pairs) is sent inside the URL of a GET request:
- GET /RegisterDao.jsp?name1=value1&name2=value2

## Anatomy of Post Request

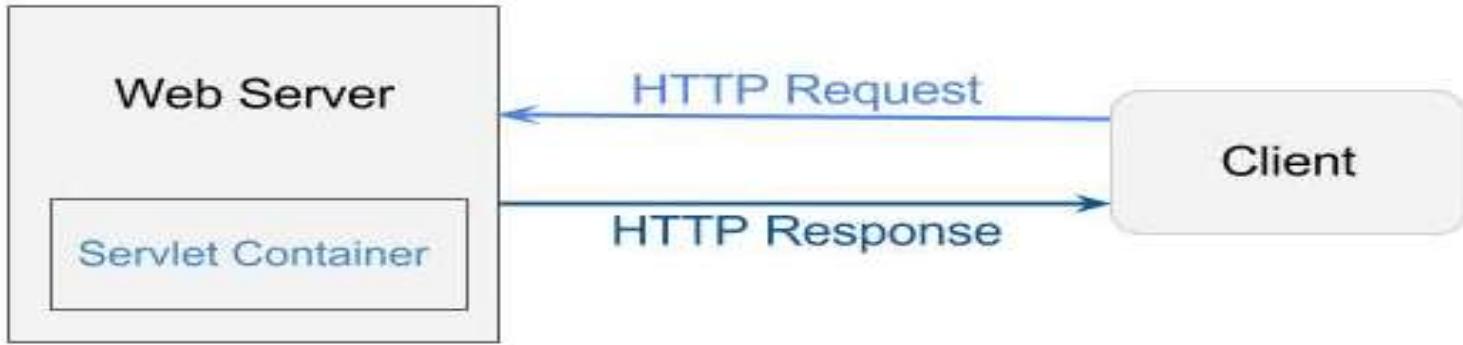
- The query string (name/value pairs) is sent in HTTP message body for a POST request:
- POST/RegisterDao.jsp HTTP/1.1
- Host: www.javatpoint.com
- name1=value1&name2=value2

The HTTP Method	Path to the source on Web Server	Parameters to the server	Protocol Version Browser supports
GET	/RegisterDao.jsp?user=ravi&pass=java		HTTP/1.1
The Request Headers	Host: www.javatpoint.com User-Agent: Mozilla/5.0 Accept-text/xml,text/html,text/plain,image/jpeg Accept-Language: en-us,en Accept-Encoding: gzip,deflate Accept-Charset: ISO-8859-1,utf-8 Keep-Alive: 300 Connection: keep-alive		

The HTTP Method	Path to the source on Web Server	Protocol Version Browser supports
Post	/RegisterDao.jsp	HTTP/1.1
The Request Headers	Host: www.javatpoint.com User-Agent: Mozilla/5.0 Accept: text/xml,text/html,text/plain,image/jpeg Accept-Language: en-us,en Accept-Encoding: gzip,deflate Accept-Charset: ISO-8859-1,utf-8 Keep-Alive:300 Connection:keep-alive User=ravi&pass=java	} Message body

# Servlet Container

- It provides the runtime environment for JavaEE (j2ee) applications. The client/user can request only a static WebPages from the server. If the user wants to read the web pages as per input then the servlet container is used in java.
- The servlet container is used in java for dynamically generate the web pages on the server side. Therefore the servlet container is the part of a web server that interacts with the servlet for handling the dynamic web pages from the client.



## Servlet Container States

The servlet container is the part of web server which can be run in a separate process. We can classify the servlet container states in three types:

**Standalone:** It is typical Java-based servers in which the servlet container and the web servers are the integral part of a single program. For example:- Tomcat running by itself

**In-process:** It is separated from the web server, because a different program runs within the address space of the main server as a plug-in. For example:- Tomcat running inside the JBoss.

**Out-of-process:** The web server and servlet container are different programs which are run in a different process. For performing the communications between them, web server uses the plug-in provided by the servlet container.

# Servlet API

- The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.
- The **javax.servlet** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.
- The **javax.servlet.http** package contains interfaces and classes that are responsible for http requests only.

# Interfaces in javax.servlet package

- There are many interfaces in javax.servlet package. They are as follows:

1. **Servlet**
2. **ServletRequest**
3. **ServletResponse**
4. **RequestDispatcher**
5. **ServletConfig**
6. **ServletContext**
7. **SingleThreadModel**
8. **Filter**
9. **FilterConfig**
10. **FilterChain**
11. **ServletRequestListener**
12. **ServletRequestAttributeListener**
13. **ServletContextListener**
14. **ServletContextAttributeListener**

# Classes in javax.servlet package

- There are many classes in javax.servlet package. They are as follows:
- GenericServlet
- ServletInputStream
- ServletOutputStream
- ServletRequestWrapper
- ServletResponseWrapper
- ServletRequestEvent
- ServletContextEvent
- ServletRequestAttributeEvent
- ServletContextAttributeEvent
- ServletException
- UnavailableException

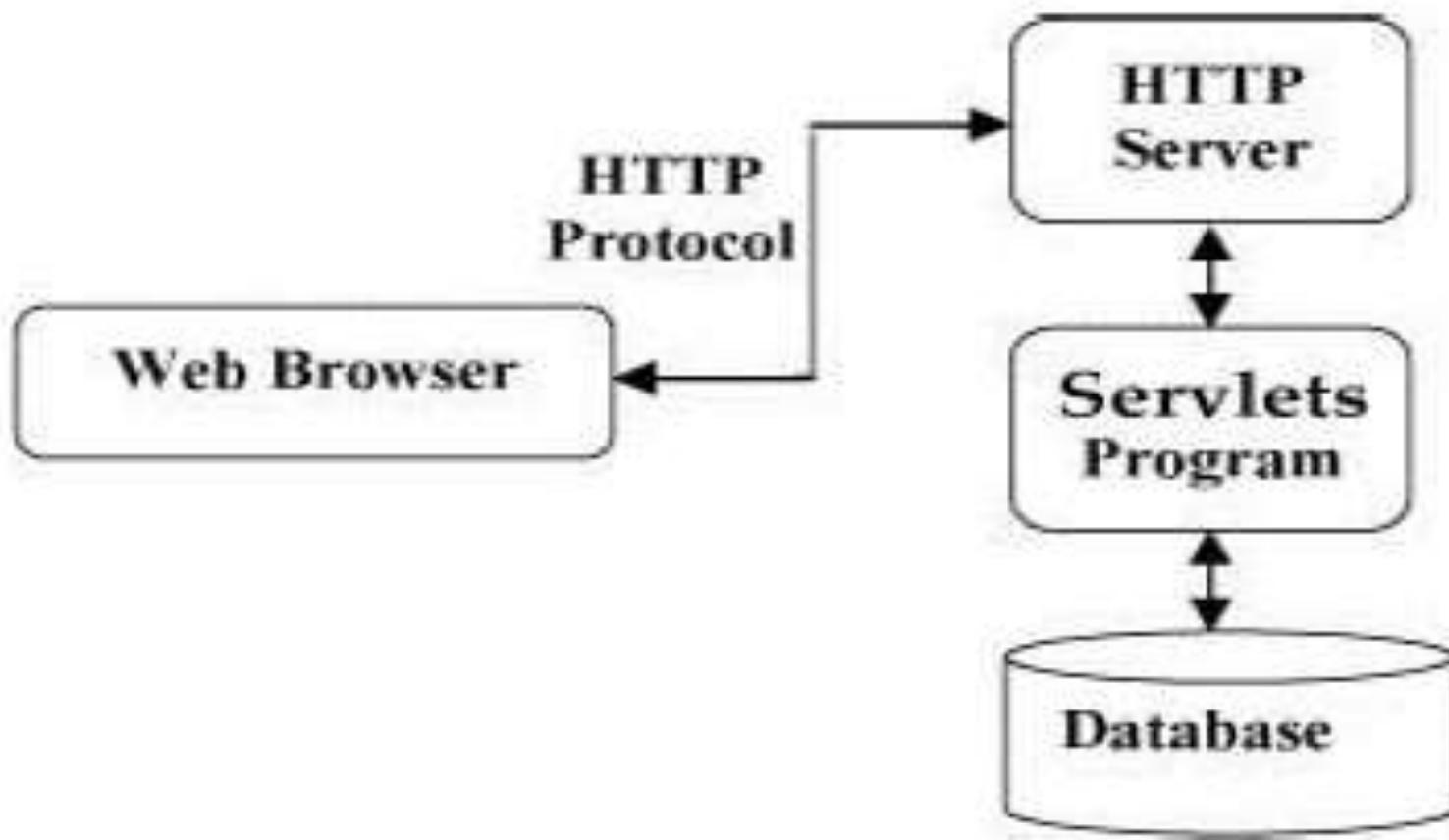
# Interfaces in javax.servlet.http package

- There are many interfaces in javax.servlet.http package. They are as follows:
  - HttpServletRequest
  - HttpServletResponse
  - HttpSession
  - HttpSessionListener
  - HttpSessionAttributeListener
  - HttpSessionBindingListener
  - HttpSessionActivationListener
  - HttpSessionContext (deprecated now)

# Classes in javax.servlet.http package

- There are many classes in javax.servlet.http package. They are as follows:
- HttpServlet
- Cookie
- HttpServletRequestWrapper
- HttpServletResponseWrapper
- HttpSessionEvent
- HttpSessionBindingEvent
- HttpUtils (deprecated now)

# Servlets Architecture:



# Servlets Tasks:

Servlets perform the following major tasks:

1. Read the explicit data sent by the clients (browsers). This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.
2. Read the implicit HTTP request data sent by the clients (browsers). This includes cookies, media types and compression schemes the browser understands, and so forth.
3. Process the data and generate the results.
4. This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.

- Send the explicit data (i.e., the document) to the clients (browsers).
- This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), Excel, etc.
- Send the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

# Servlet Interface

- **Servlet interface** provides common behaviour to all the servlets.
- Servlet interface needs to be implemented for creating any servlet (either directly or indirectly).
- It provides 3 life cycle methods that are used
  - 1.to initialize the servlet,
  2. to service the requests, and
  - 3.to destroy the servlet and 2 non-life cycle methods.

# Methods of Servlet interface

- There are 5 methods in Servlet interface. The init, service and destroy are the life cycle methods of servlet. These are invoked by the web container.

Method	Description
<b>public void init(ServletConfig config)</b>	initializes the servlet. It is the life cycle method of servlet and invoked by the web container only once.
<b>public void service(ServletRequest request,ServletResponse response)</b>	provides response for the incoming request. It is invoked at each request by the web container.
<b>public void destroy()</b>	It is invoked only once and indicates that servlet is being destroyed.
<b>public ServletConfig getServletConfig()</b>	returns the object of ServletConfig.
<b>public String getServletInfo()</b>	returns information about servlet such as writer, copyright, version etc.

# Servlet Example by implementing Servlet interface

```
import java.io.*;
import javax.servlet.*;
public class First implements Servlet{
    ServletConfig config=null;
    public void init(ServletConfig config){
        this.config=config;
        System.out.println("servlet is initialized");
    }
    public void service(ServletRequest req,ServletResponse res)
        throws IOException,ServletException{
        res.setContentType("text/html");
        PrintWriter out=res.getWriter();
        out.print("<html><body>");
        out.print("<b>hello simple servlet</b>");
        out.print("</body></html>");
    }
}
```

```
}

public void destroy(){System.out.println("servlet is destroyed");}

public ServletConfig getServletConfig(){return config;}

public String getServletInfo(){return "copyright 2007-1010";}

}
```

# GenericServlet class

- **GenericServlet** class implements **Servlet**, **ServletConfig** and **Serializable** interfaces. It provides the implementation of all the methods of these interfaces except the service method.
- GenericServlet class can handle any type of request so it is protocol-independent.

# Methods of GenericServlet class

- There are many methods in GenericServlet class. They are as follows:
- **public void init(ServletConfig config)** is used to initialize the servlet.
- **public abstract void service(ServletRequest request, ServletResponse response)** provides service for the incoming request. It is invoked at each time when user requests for a servlet.
- **public void destroy()** is invoked only once throughout the life cycle and indicates that servlet is being destroyed.
- **public ServletConfig getServletConfig()** returns the object of ServletConfig.
- **public String getServletInfo()** returns information about servlet such as writer, copyright, version etc.
- **public void init()** it is a convenient method for the servlet programmers, now there is no need to call super.init(config)

# HttpServlet class

- The HttpServlet class extends the GenericServlet class and implements Serializable interface. It provides http specific methods such as doGet, doPost, doHead, doTrace etc.

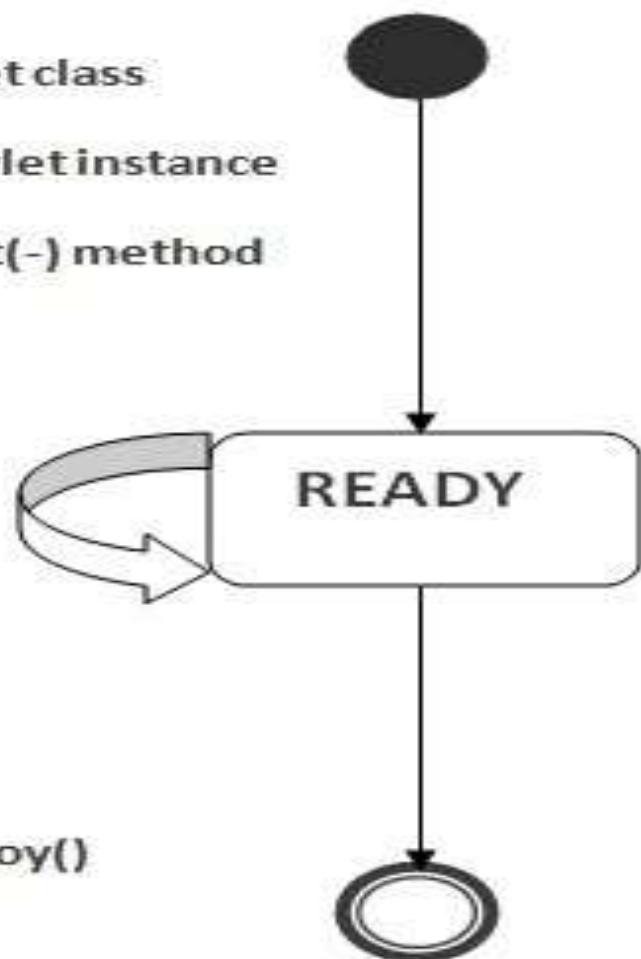
# Methods of HttpServlet class

- There are many methods in HttpServlet class. They are as follows:
- **public void service(ServletRequest req,ServletResponse res)** dispatches the request to the protected service method by converting the request and response object into http type.
- **protected void service(HttpServletRequest req, HttpServletResponse res)** receives the request from the service method, and dispatches the request to the doXXX() method depending on the incoming http request type.
- **protected void doGet(HttpServletRequest req, HttpServletResponse res)** handles the GET request. It is invoked by the web container.
- **protected void doPost(HttpServletRequest req, HttpServletResponse res)** handles the POST request. It is invoked by the web container.
- **protected void doHead(HttpServletRequest req, HttpServletResponse res)** handles the HEAD request. It is invoked by the web container.
- **protected void doOptions(HttpServletRequest req, HttpServletResponse res)** handles the OPTIONS request. It is invoked by the web container.

# Life Cycle of a Servlet (Servlet Life Cycle)

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.

- 1.Load servlet class
- 2.Create servlet instance
- 3.Call the init(-) method
- 4.Call the service(-,-)  
method
- 5.Call the destroy()  
method



## 1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

## 2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

## 3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below:

**public void init(ServletConfig config) throws ServletException**

- 4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

- **public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException**

## 5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

- **public void destroy()**

# GET Method and POST Method.

- when you need to pass some information from your browser to web server and ultimately to your backend program.
- The browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

- **GET**
- The GET type request is normally used for simple HTML page requests. It has this syntax:

```
public void doGet(HttpServletRequest request,  
HttpServletResponse response) throws  
IOException, ServletException { //your code here}
```

- **POST**
- The POST type request is most often used by HTML forms. It has this syntax:

```
public void doPost(HttpServletRequest request,  
HttpServletResponse response) throws  
IOException, ServletException { //your code here}
```

# Form-Data Submission Methods: GET | POST

- Two request methods, GET and POST, are available for submitting form data, to be specified in the <form>'s attribute "method=GET | POST".
- GET and POST performs the same basic function.
- However, in a GET request, the query string is appended **behind the URL, separated by a '?'**. Whereas in a POST request, the query string is kept in the **request body (and not shown in the URL)**.

- The length of query string in a GET request is limited by the maximum length of URL permitted, whereas it is unlimited in a POST request.
- I recommend POST request for production.
- To try out the POST request, modify the "form\_input.html":
  - <form **method="post"** action="echo">
  - .....
  - </form>

```
public class MyServlet extends HttpServlet {  
    // doGet() handles GET request  
    @Override public void doGet(HttpServletRequest request,  
        HttpServletResponse response)  
        throws IOException, ServletException  
{  
    .....  
    .....  
}  
// doPost() handles POST request  
@Override public void doPost(HttpServletRequest request,  
    HttpServletResponse response)  
throws IOException, ServletException  
{  
    doGet(request, response);  
    // call doGet()  
}  
}
```

# Request Header and Response Header

- HTTP is a request-response protocol.
- The client sends a request message to the server.
- The server, in turn, returns a response message.
- The request and response messages consists of two parts: header (information about the message) and body (contents).
- Header provides information about the messages. The data in header is organized in name-value pairs.
- Read "[HTTP Request and Response Messages](#)" for the format, syntax of request and response messages.

# HttpServletRequest

- The request message is encapsulated in an HttpServletRequest object, which is passed into the doGet() methods.
- HttpServletRequest provides many methods for you to retrieve the headers:

## General methods:

- `getHeader(name)`, `getHeaders(name)`, `getHeaderNames()`.

## Specific methods:

- `getContentLength()`, `getContentType()`, `getCookies()`, `getAuthType()`, etc.

## URL related:

- `getRequestURI()`, `getQueryString()`, `getProtocol()`, `getMethod()`.

# HttpServletResponse

- The response message is encapsulated in the HttpServletResponse, which is passed into doGet() by reference for receiving the servlet output.
- setStatusCode(int statuscode), sendError(int code, String message), sendRedirect(url).
- response.setHeader(String headerName, String headerValue).
- setContentType(String mimeType), setContentLength(int length), etc.

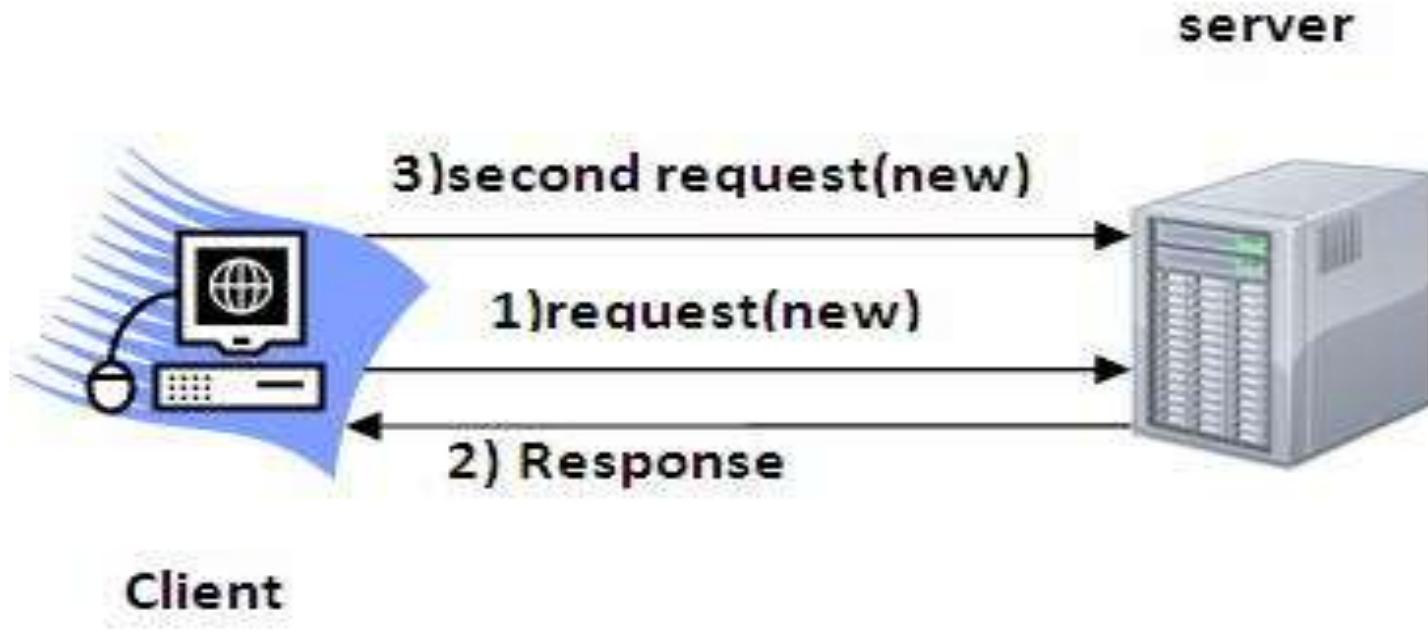
# Session Tracking

- HTTP is a *stateless* protocol.
- In other words, the current request does not know what has been done in the previous requests.
- This creates a problem for applications that runs over many requests, such as online shopping (or shopping cart). You need to maintain a so-called *session* to pass data among the multiple requests.
- You can maintain a session via one of these three approaches:
- **Cookie**: A cookie is a small text file that is stored in the client's machine, which will be send to the server on each request. You can put your session data inside the cookie. The biggest problem in using cookie is clients may disable the cookie.

- **URL Rewriting:** Passes data by appending a short text string at the end of every URL, e.g., `http://host/path/file.html;jsessionid=123456`. You need to rewrite all the URLs (e.g., the "action" attribute of `<form>`) to include the session data.
- **Hidden field in an HTML form:** pass data by using hidden field tag (`<input type="hidden" name="session" value="...." />`). Again, you need to include the hidden field in all the pages.

# Session Tracking in Servlets

- **Session** simply means a particular interval of time.
- **Session Tracking** is a way to maintain state (data) of an user. It is also known as **session management** in servlet.
- Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.



Why use Session Tracking?

To recognize the user It is used to recognize the particular user.

# Session Tracking Techniques

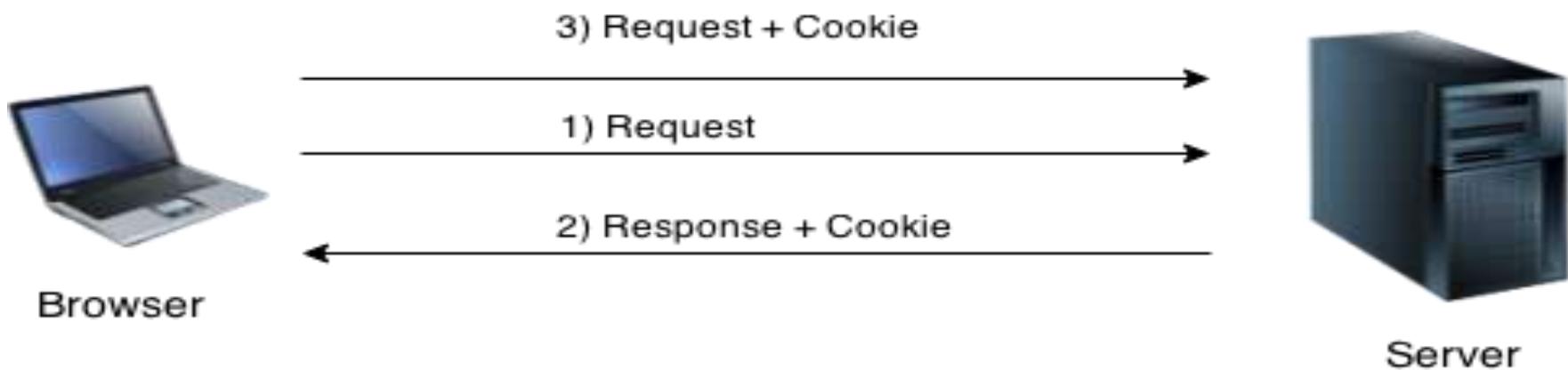
- There are four techniques used in Session tracking:
- **Cookies**
- **Hidden Form Field**
- **URL Rewriting**
- **HttpSession**

# Cookies in Servlet

- A **cookie** is a small piece of information that is persisted between the multiple client requests.
- A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

# How Cookie works

- By default, each request is considered as a new request.
- In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.



# Types of Cookie

- There are 2 types of cookies in servlets.
  1. Non-persistent cookie
  2. Persistent cookie
- **Non-persistent cookie**
- It is **valid for single session** only. It is removed each time when user closes the browser.
- **Persistent cookie**
- It is **valid for multiple session** . It is not removed each time when user closes the browser. It is removed only if user logout or signout.

## Advantage of Cookies

- Simplest technique of maintaining the state.
- Cookies are maintained at client side.

## Disadvantage of Cookies

- It will not work if cookie is disabled from the browser.
- Only textual information can be set in Cookie object.

## Cookie class

**javax.servlet.http.Cookie** class provides the functionality of using cookies.

It provides a lot of useful methods for cookies.

- Constructor of Cookie class

Constructor	Description
Cookie()	constructs a cookie.
Cookie(String name, String value)	constructs a cookie with a specified name and value.

# Useful Methods of Cookie class

Method	Description
public void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.
public String getName()	Returns the name of the cookie. The name cannot be changed after creation.
public String getValue()	Returns the value of the cookie.
public void setName(String name)	changes the name of the cookie.
public void setValue(String value)	changes the value of the cookie.

- How to create Cookie?

Let's see the simple code to create cookie.

```
Cookie ck=new Cookie("user","sonoo jaiswal");//creating cookie object  
response.addCookie(ck);//adding cookie in the response
```

- How to delete Cookie?

Let's see the simple code to delete cookie. It is mainly used to logout or signout the user.

```
Cookie ck=new Cookie("user","");
ck.setMaxAge(0);
response.addCookie(ck);
```

# How to get Cookies?

Let's see the simple code to get all the cookies.

```
Cookie ck[]=request.getCookies();
for(int i=0;i<ck.length;i++){
    out.print("<br>" + ck[i].getName() + " " + ck[i].getValue()
()); //printing name and value of cookie
}
```

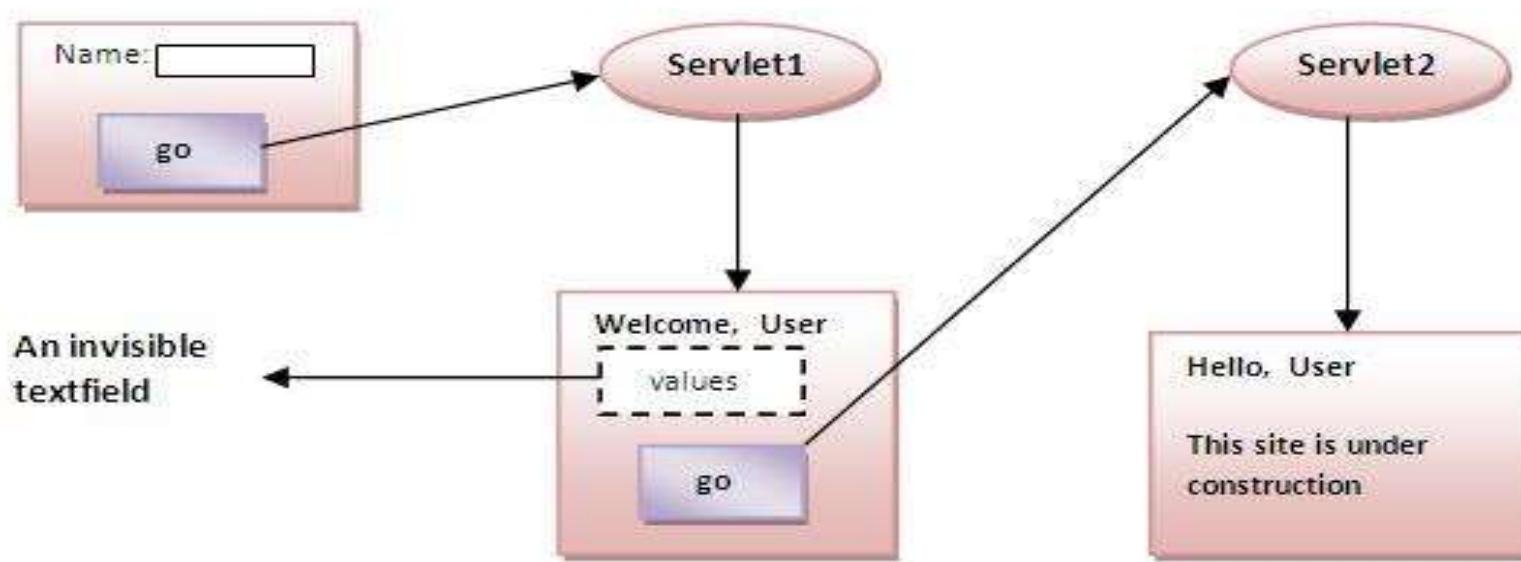
## 2) Hidden Form Field

- In case of Hidden Form Field a **hidden (invisible) textfield** is used for maintaining the state of an user.
- In such case, we store the information in the hidden field and get it from another servlet. This approach is better if we have to submit form in all the pages and we don't want to depend on the browser.
- `<input type="hidden" name="uname" value="Vimal Jaiswal">`
- Here, uname is the hidden field name and Vimal Jaiswal is the hidden field value.

- **Advantage of Hidden Form Field**
- It will always work whether cookie is disabled or not.
- **Disadvantage of Hidden Form Field:**
- It is maintained at server side.
- Extra form submission is required on each pages.
- Only textual information can be used.

## Example of using Hidden Form Field

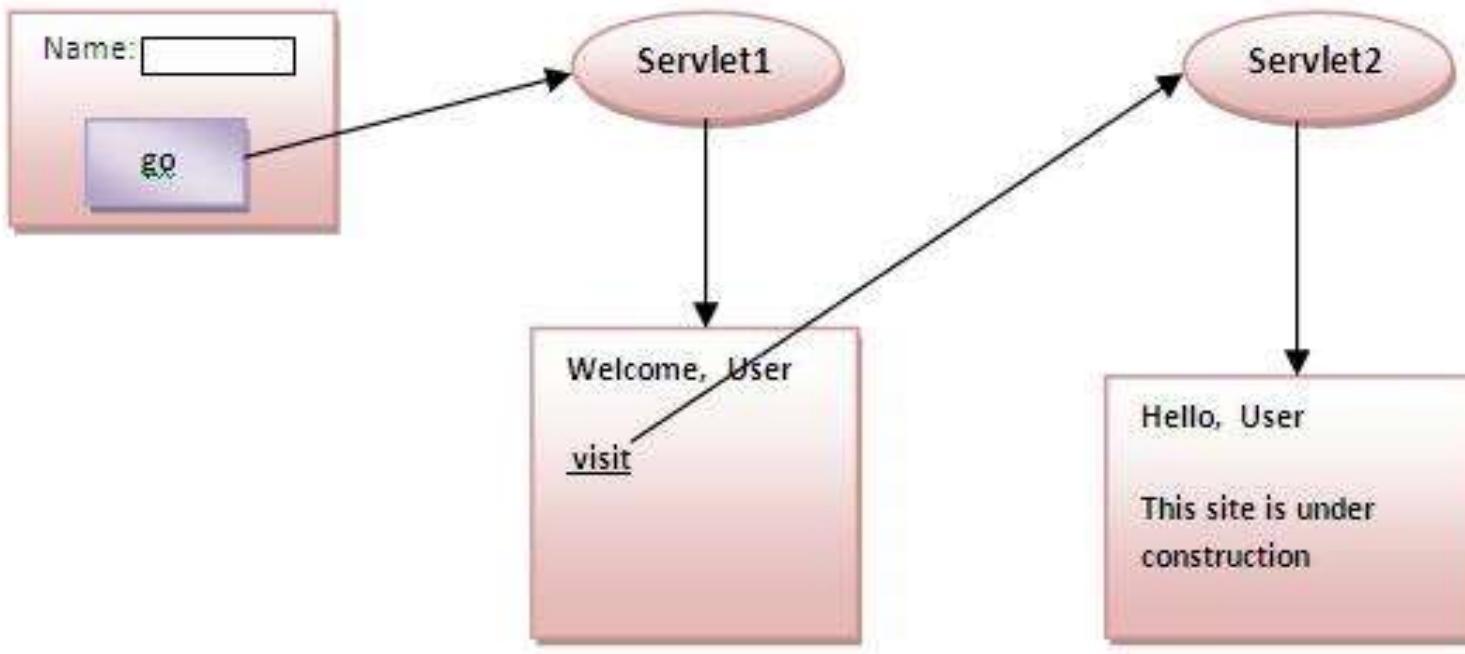
In this example, we are storing the name of the user in a hidden textfield and getting that value from another servlet.



```
<form action="servlet1">  
Name:<input type="text" name="userNmae"/><br/>  
<input type="submit" value="go"/>  
</form>
```

### 3)URL Rewriting

- In URL rewriting, we append a token or identifier to the URL of the next Servlet or the next resource. We can send parameter name/value pairs using the following format:
- **url?name1=value1&name2=value2&??**
- A name and a value is separated using an equal = sign, a parameter name/value pair is separated from another parameter using the ampersand(&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server.
- From a Servlet, we can use `getParameter()` method to obtain a parameter value.



## Advantage of URL Rewriting

It will always work whether cookie is disabled or not (browser independent).

Extra form submission is not required on each pages.

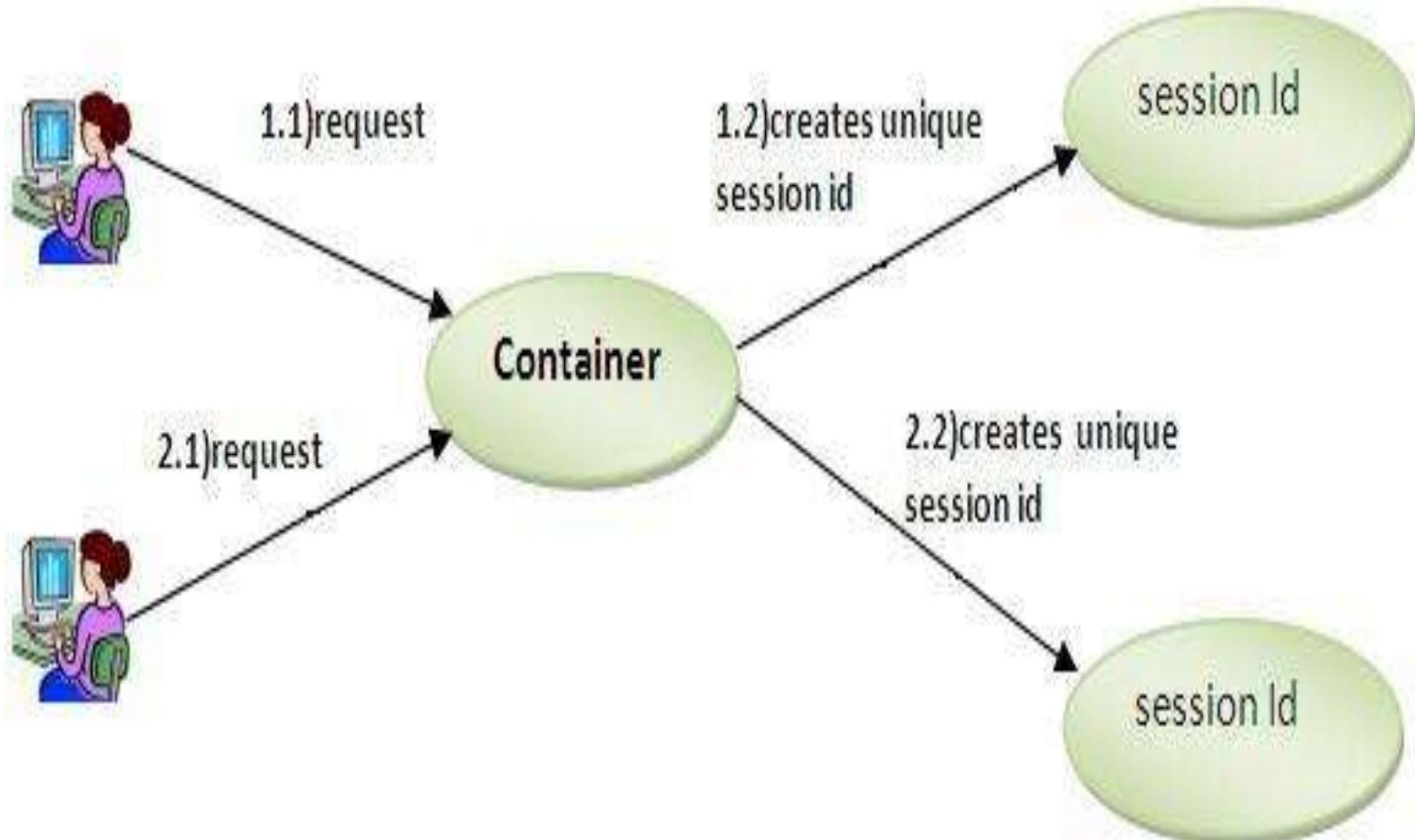
## Disadvantage of URL Rewriting

It will work only with links.

It can send Only textual information.

## 4) HttpSession interface

- In such case, container creates a session id for each user. The container uses this id to identify the particular user. An object of HttpSession can be used to perform two tasks:
  - 1.bind objects
  - 2.view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.



# How to get the HttpSession object ?

- The HttpServletRequest interface provides two methods to get the object of HttpSession:
- **public HttpSession getSession():**Returns the current session associated with this request, or if the request does not have a session, creates one.
- **public HttpSession getSession(boolean create):**Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

- Commonly used methods of HttpSession interface
- **public String getId():**Returns a string containing the unique identifier value.
- **public long getCreationTime():**Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
- **public long getLastAccessedTime():**Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
- **public void invalidate():**Invalidates this session then unbinds any objects bound to it.

# JDBC

- **JAVA DATABASE CONNECTIVITY**
  1. IMPORT THE PACKAGE(JAVA.SQL.\* )
  2. LOAD & REGISTER THE DRIVER
  3. ESTABLISH THE CONNECTION---CREATE AN OBJECT
  4. CREATE THE STATEMENT
    - STATEMENT
    - PREPARED STATEMENT
    - CALLABLE STATEMENT
  5. EXECUTE THE QUERY
  6. PROCESS THE RESULT
  7. CLOSE DATABASE

# EXAMPLE

```
Import java.sql.*;  
Public static void main()  
{ class.forName("con.mysql.jdbc.driver");  
Connection con=Driver  
    manager.getConnection("url","uid","pwd");  
Statement st= con.create Statement();  
ResultSet=st.executeQuery("Select * from  
student");  
while(rs.next())  
System.out.println(rs.getInt(1)+" "+rs.Student(2})  
St.close();con.close();
```

# Java JDBC



- Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.
- Why use JDBC?
- What is API ?
- API (Application programming interface) is a document that contains description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc

# Introduction

JDBC stands for **Java Database Connectivity**, which is a standard Java API for database-independent connectivity between the **Java programming language and a wide range of databases**.

The JDBC library includes APIs for each of the tasks commonly associated with database usage:

- Making a connection to a database
- Creating SQL or MySQL statements
- Executing that SQL or MySQL queries in the database
- Viewing & Modifying the resulting records

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as:

Java Applications  
Java Applets  
Java Servlets  
Java Server Pages (JSPs)  
Enterprise JavaBeans (EJBs)

- All of these different executables are able to use a JDBC driver to access a database and take advantage of the stored data.
- JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

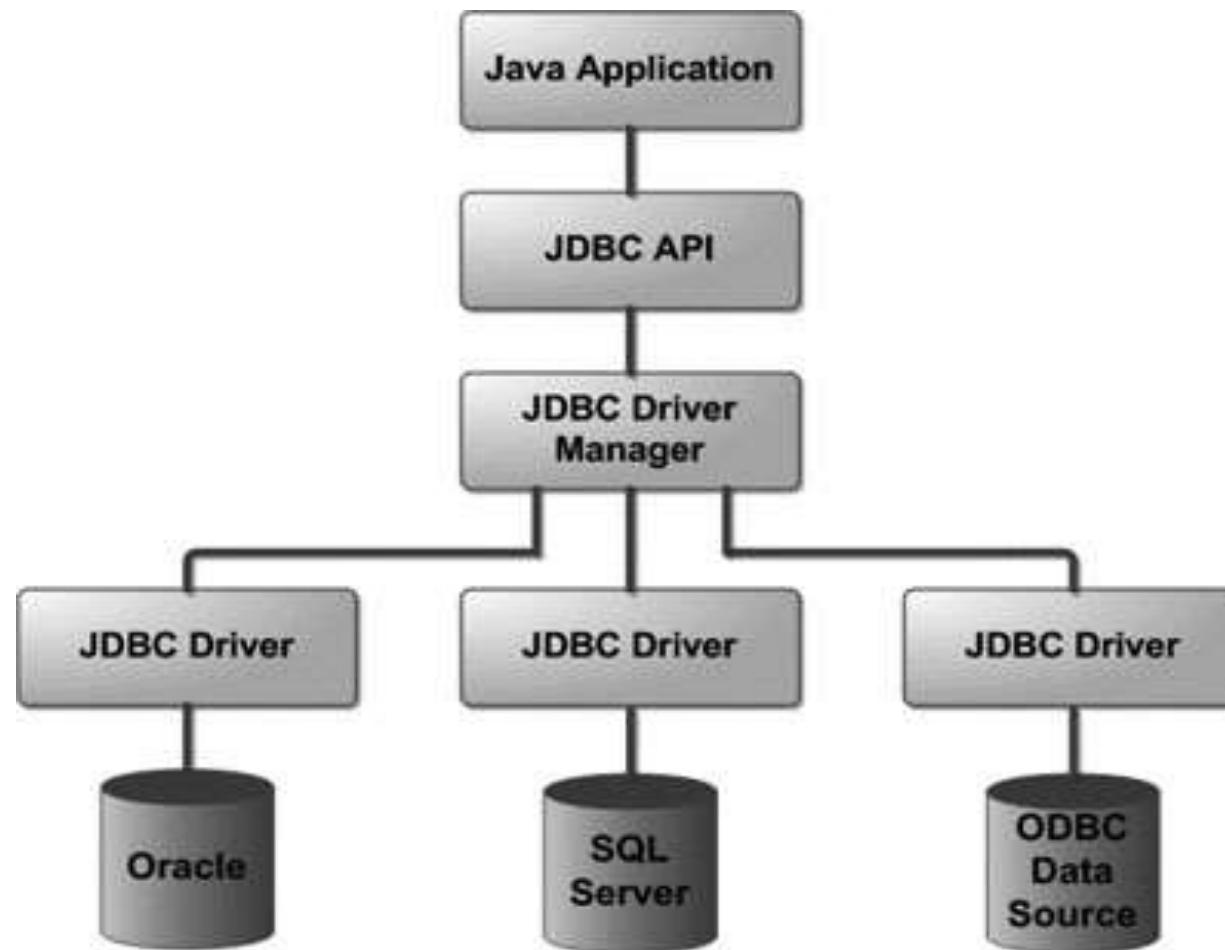
**Cont..**

## JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access but in general JDBC Architecture consists of two layers:

- **JDBC API:** This provides the application-to-JDBC Manager connection. (`java.sql` / `javax.sql`)
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application:



## Common JDBC Components:

The JDBC API provides the following interfaces and classes:

1. **DriverManager:** This class manages a **list of database drivers**. Matches connection requests from the java application with the **proper database driver** using communication sub protocol. The first driver that recognizes a certain sub protocol under JDBC will be used to establish a database Connection.
2. **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use **DriverManager objects**, which manage objects of this type. It also abstracts the details associated with working with Driver objects.
3. **Connection:** This interface provides all methods **for contacting a database**. The connection object represents communication context, i.e., all communication with database is through connection object only.

4. **Statement:** You use objects created from this interface to submit the SQL statements to the database.
5. **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
6. **SQLException:** This class handles any errors that occur in a database application.

# 5 Steps to connect to the database in java

- There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:
- Register the driver class
- Creating connection
- Creating statement
- Executing queries
- Closing connection

## 1) Register the driver class

The `forName()` method of `Class` class is used to register the driver class. This method is used to dynamically load the driver class.

### Syntax of `forName()` method

**public static void forName(String className) throws  
ClassNotFoundException**

### Example to register the `OracleDriver` class

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

## 2) Create the connection object

The `getConnection()` method of `DriverManager` class is used to establish connection with the database.

### Syntax of `getConnection()` method

1) **public static Connection getConnection(String url)**

**throws SQLException**

2) **public static Connection getConnection(String url,**

**String name, String password) throws SQLException**

### Example to establish connection with the Oracle database

```
Connection con=DriverManager.getConnection  
("jdbc:oracle:thin:@localhost:1521:xe","system","pass  
word");
```

### 3) Create the Statement object

The `createStatement()` method of `Connection` interface is used to create statement. The object of statement is responsible to execute queries with the database.

#### Syntax of `createStatement()` method

**public Statement createStatement()throws SQLException**

#### Example to create the statement object

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

## 4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

### Syntax of executeQuery() method

```
public ResultSet executeQuery(String sql) throws  
SQLException
```

### Example to execute query

```
ResultSet rs=stmt.executeQuery("select * from emp");  
while(rs.next())  
{  
System.out.println(rs.getInt(1)+" "+rs.getString(2));  
}
```

## 5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

### Syntax of close() method

**public void close()throws SQLException**

### Example to close connection

```
con.close();
```

# Example to connect to the Oracle database in java

**1.Driver class:** The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**.

**2.Connection URL:** The connection URL for the oracle10G database is **jdbc:oracle:thin:@localhost:1521:xe** where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the tnsnames.ora file.

**3.Username:** The default username for the oracle database is **system**.

**4.Password:** Password is given by the user at the time of installing the oracle database.

Let's first create a table in oracle database.

- **create table emp(id number(10),name varchar2(40),age number(3));**

## Example to Connect Java Application with Oracle database

In this example, system is the username and oracle is the password of the Oracle database.

```
import java.sql.*;  
class OracleCon{  
public static void main(String args[]){  
try{  
1.//step1 load the driver class  
Class.forName("oracle.jdbc.driver.OracleDriver");  
  
2.//step2 create the connection object  
Connection con=DriverManager.getConnection(  
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
```

3./step3 create the statement object

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

4./step4 execute query

```
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
```

5./step5 close the connection object

```
con.close();
```

```
}
```

```
catch(Exception e){ System.out.println(e);} }
```

```
}
```

```
}
```

- Example to connect to the mysql database in java

**1.Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.

**2.Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, you need to replace the sonoo with your database name.

**3.Username:** The default username for the mysql database is **root**.

**4.Password:** Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

`create database sonoo;`

`use sonoo;`

`create table emp(id int(10),name varchar(40),age int(3));`

```
import java.sql.*;  
class MysqlCon{  
public static void main(String args[]){  
try{  
    1.Class.forName("com.mysql.jdbc.Driver");  
    2.Connection con=DriverManager.getConnection(  
        "jdbc:mysql://localhost:3306/sonoo","root","root");  
        //here sonoo is database name, root is username and password  
    3.Statement stmt=con.createStatement();  
    4.ResultSet rs=stmt.executeQuery("select * from emp");  
    while(rs.next())  
        System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));  
    5.con.close();  
}catch(Exception e){ System.out.println(e);}  
}  
}
```

# Database Drivers

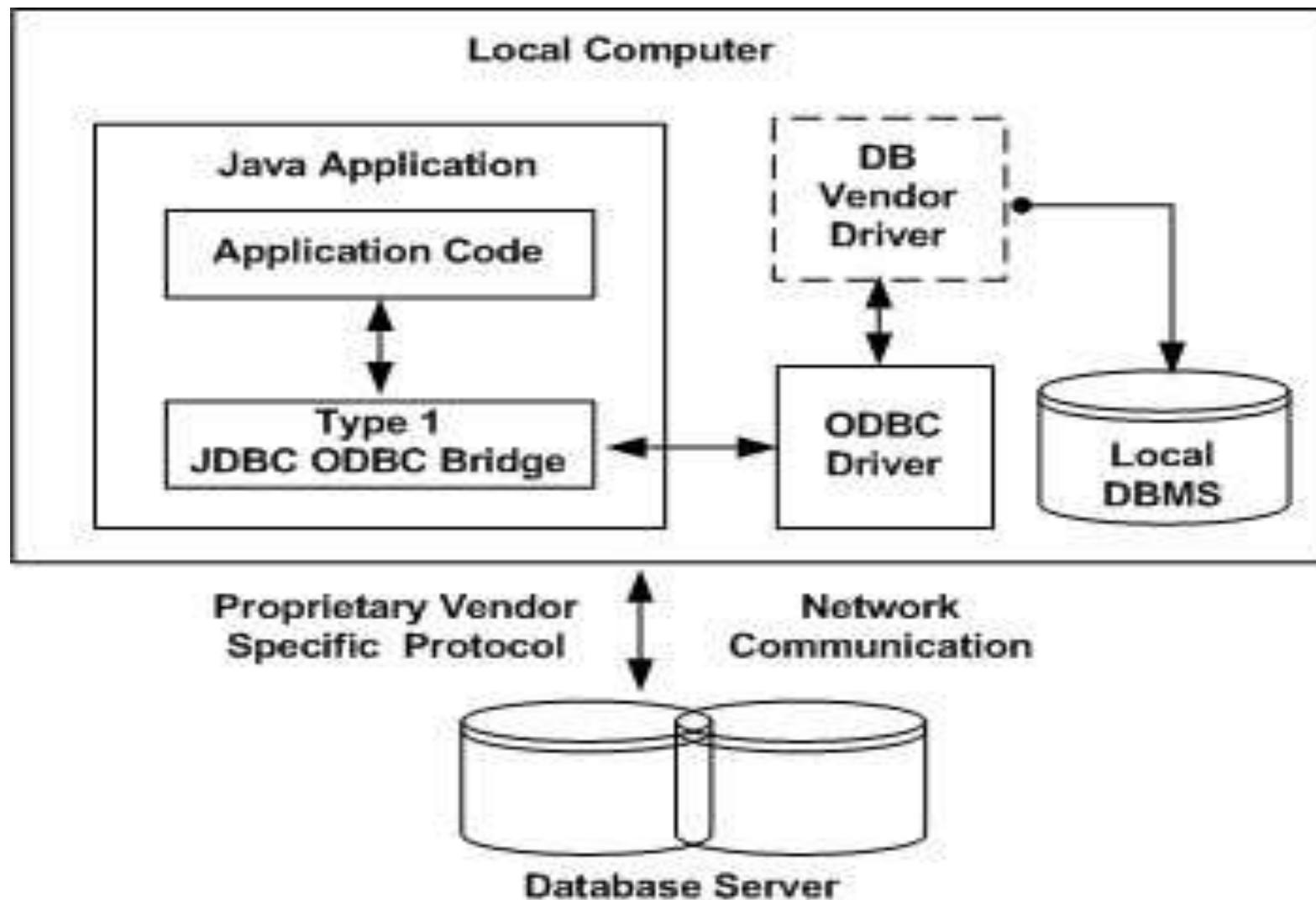
- JDBC drivers implement the defined interfaces in the JDBC API for interacting with your database server.
- For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.
- The *Java.sql* package that ships with JDK contains various classes with their behaviors defined and their actual implementations are done in third-party drivers. Third party vendors implement the *java.sql.Driver* interface in their database driver.

## JDBC Drivers Types

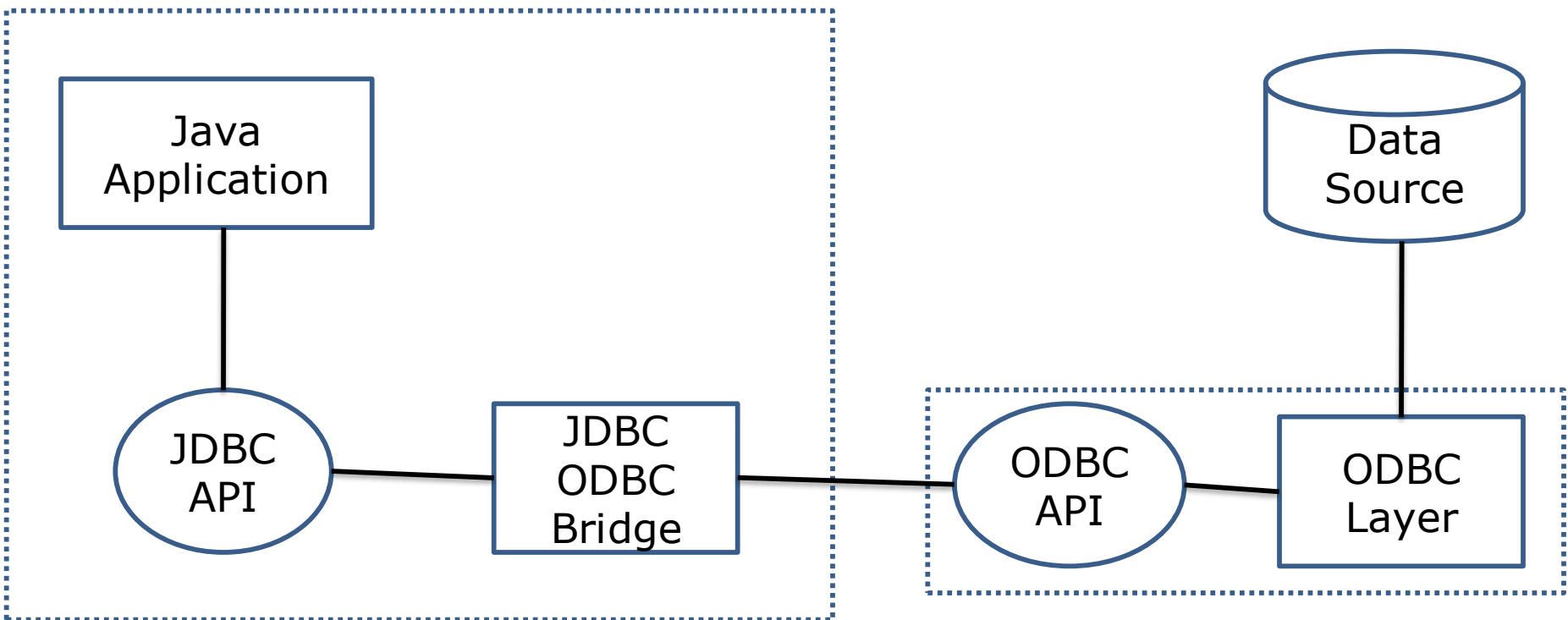
JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below:

### Type 1: JDBC-ODBC Bridge Driver:

- In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.
- When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.
- The JDBC-ODBC bridge that comes with JDK 1.2 is a good example of this kind of driver.



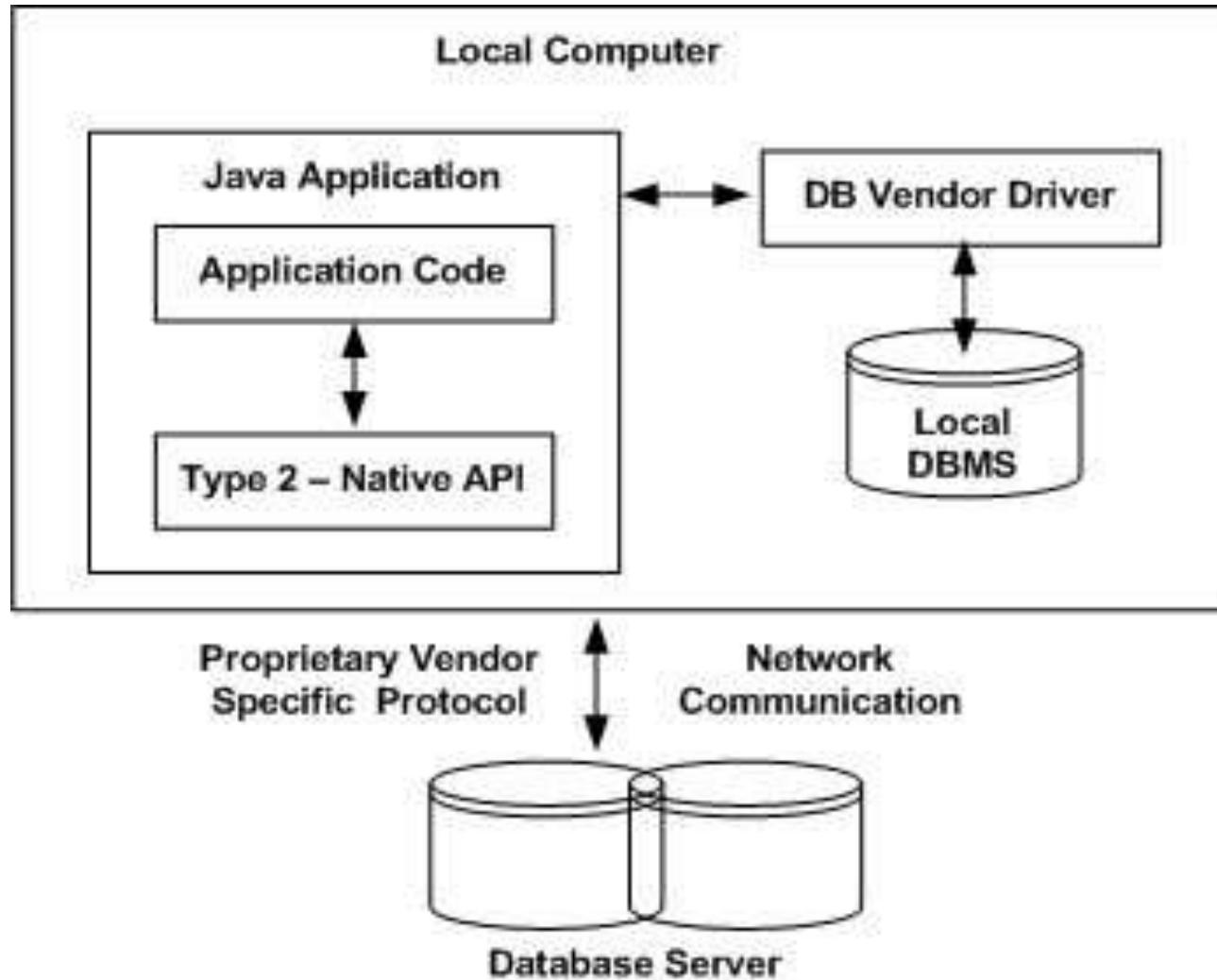
**Figure Type 1: JDBC-ODBC Bridge Driver**



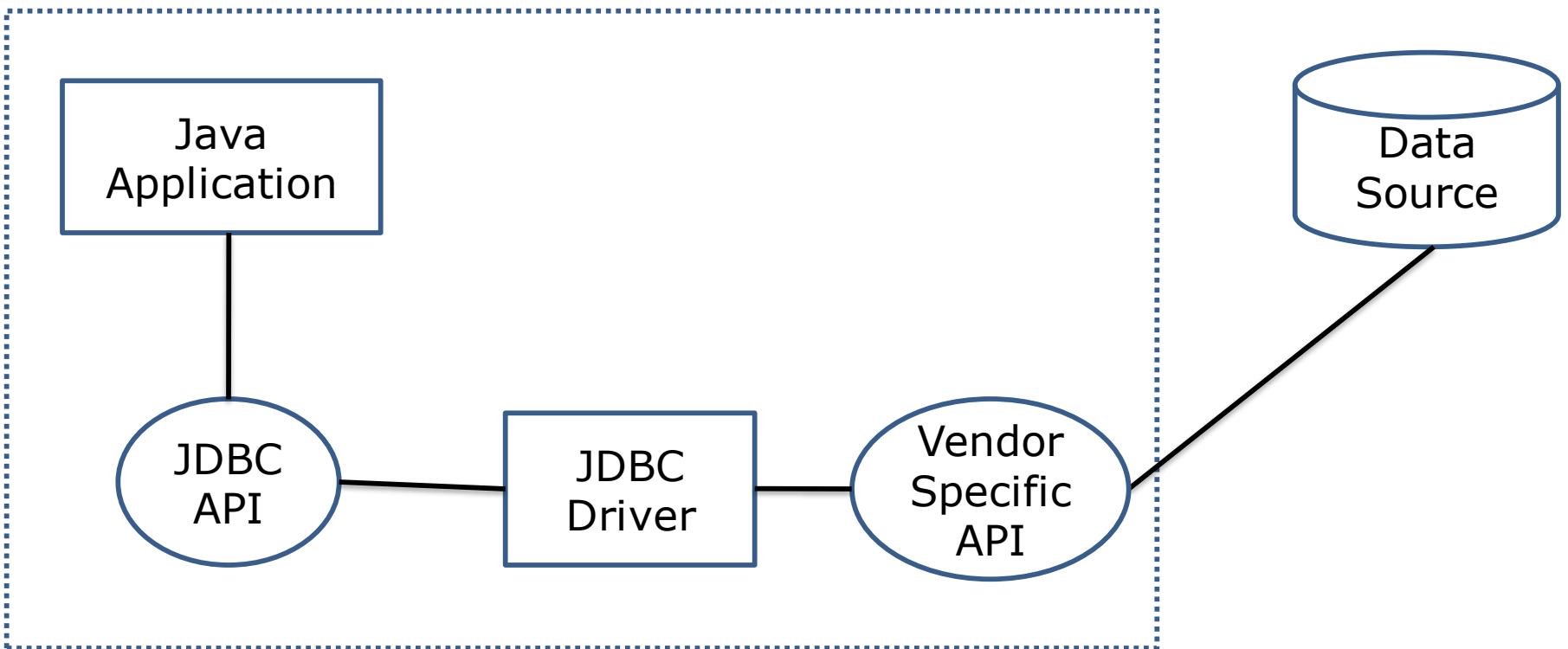
**Figure Type 1: JDBC-ODBC Bridge Driver**

## Type 2: JDBC-Native API

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database. These drivers typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine.
- If we change the Database we have to change the native API as it is specific to a database and they are mostly obsolete now but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.
- The Oracle Call Interface (OCI) diver is an example of a Type 2 driver.



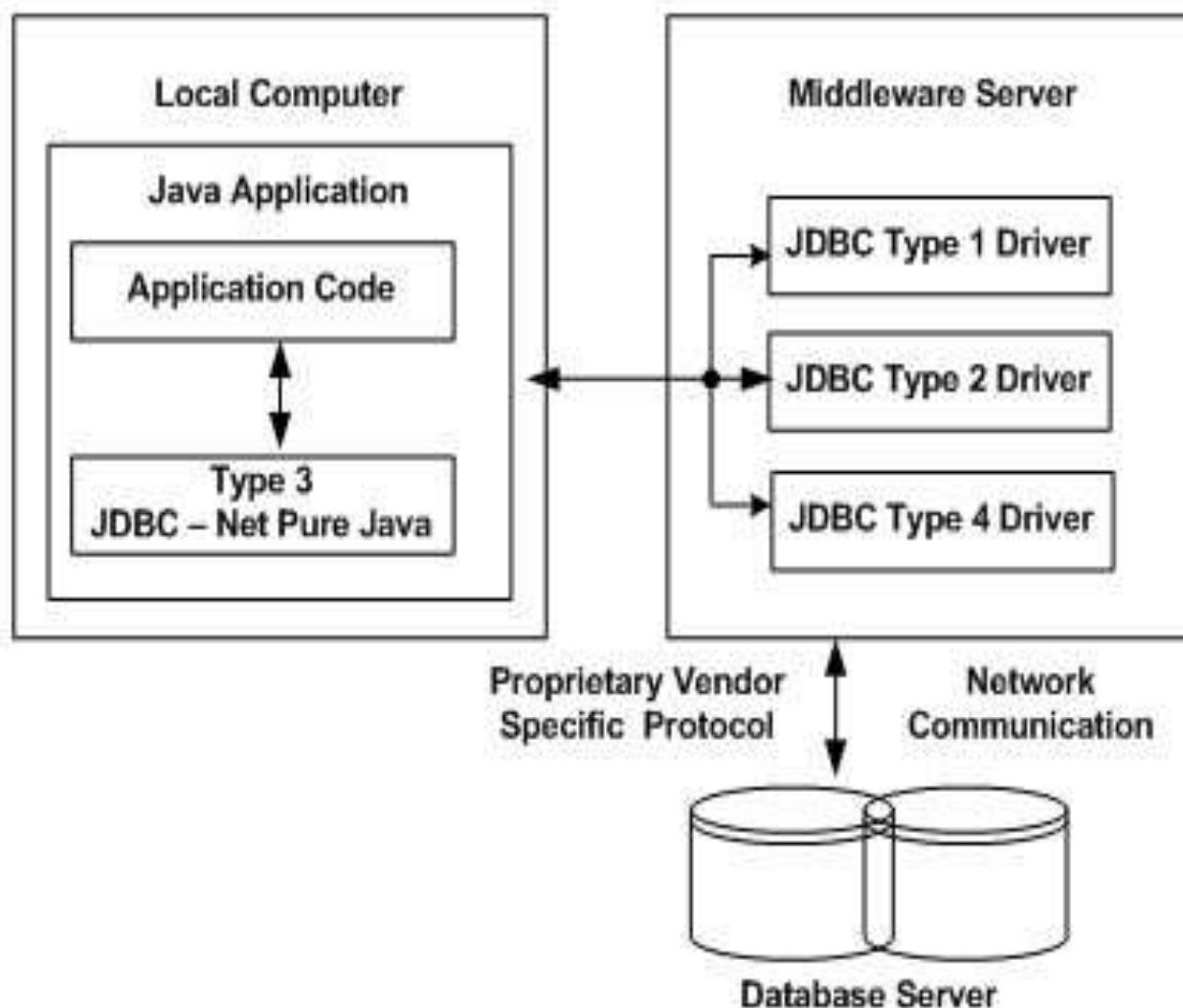
**Figure Type 2: JDBC-Native API**



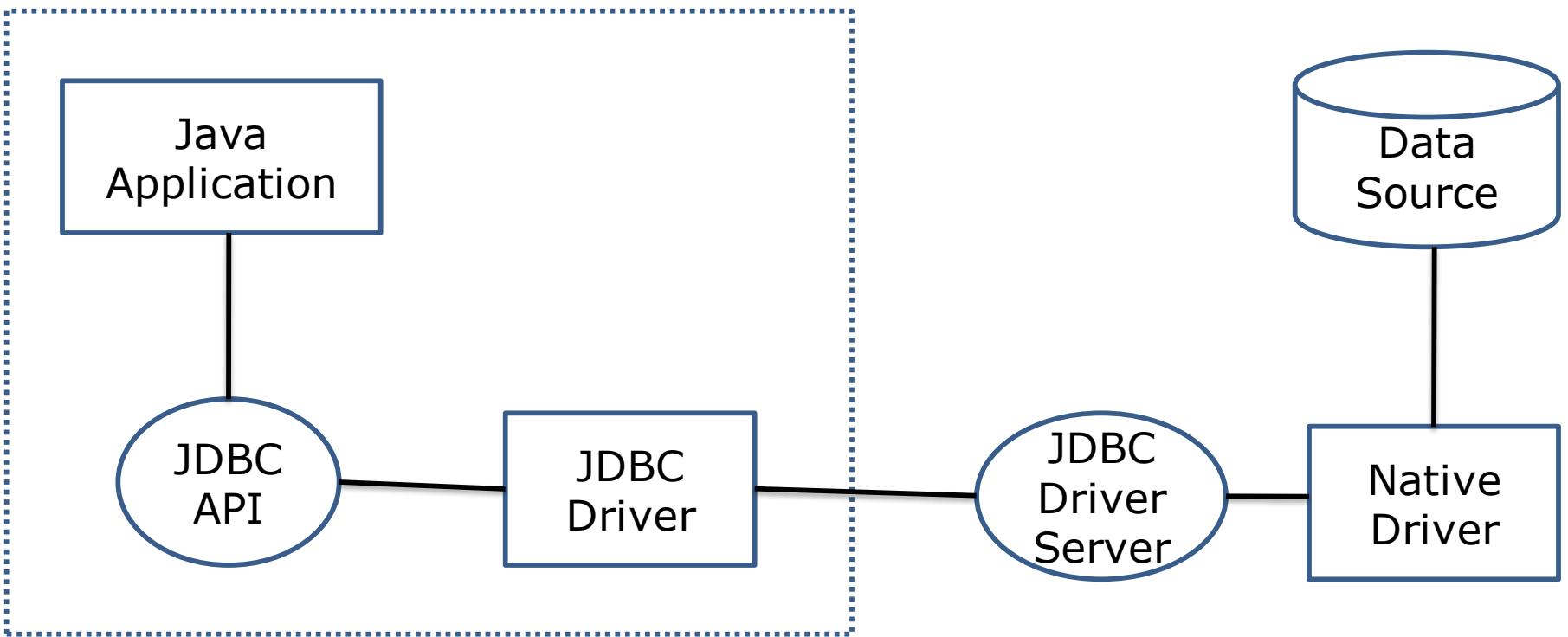
**Figure Type 2: JDBC-Native API**

## Type 3: JDBC-Part Java

- In a Type 3 driver, a three-tier approach is used to accessing databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.
- You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.
- Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.



**Figure Type 3: JDBC-Part Java**

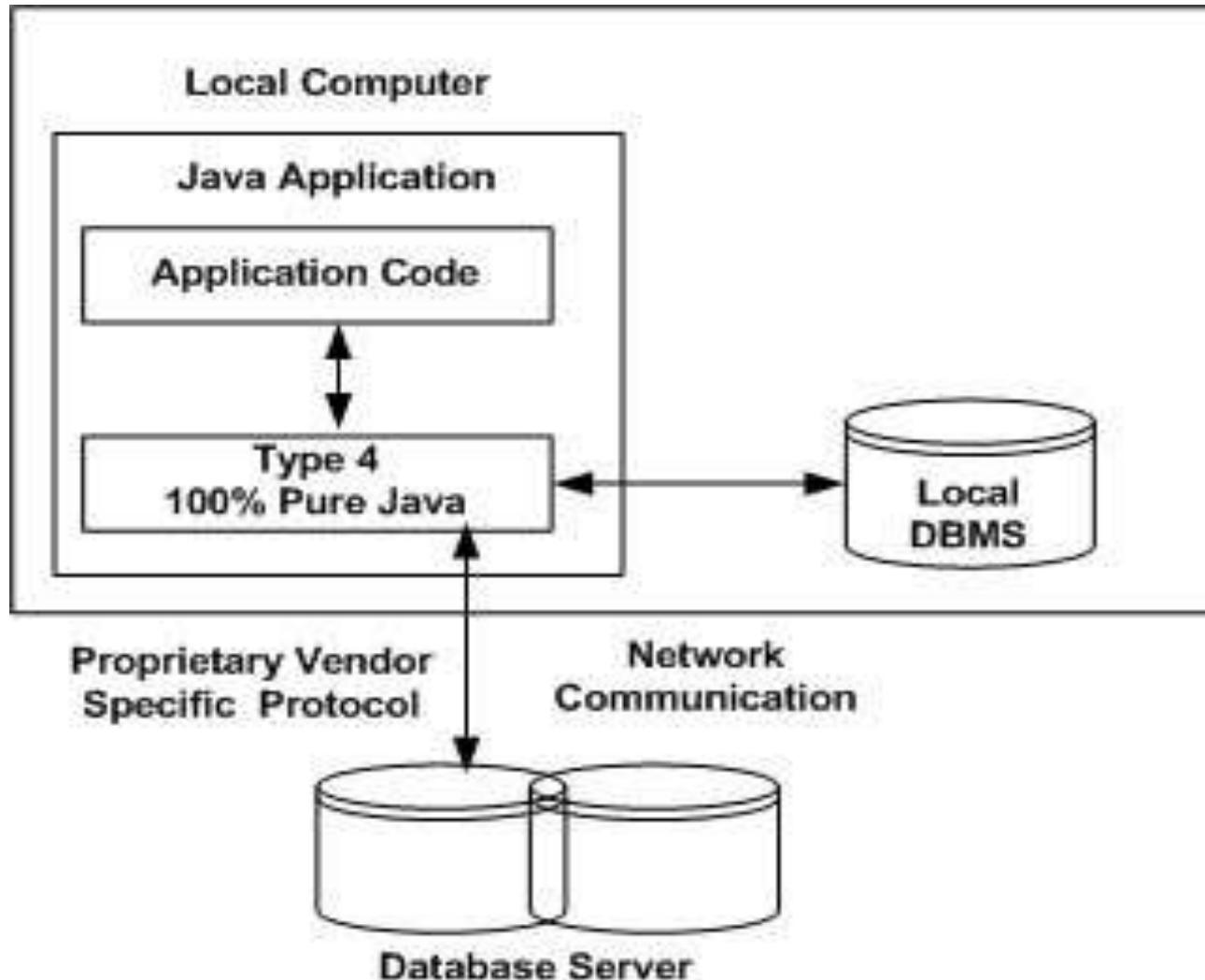


**Figure Type 3: JDBC-Type Java**

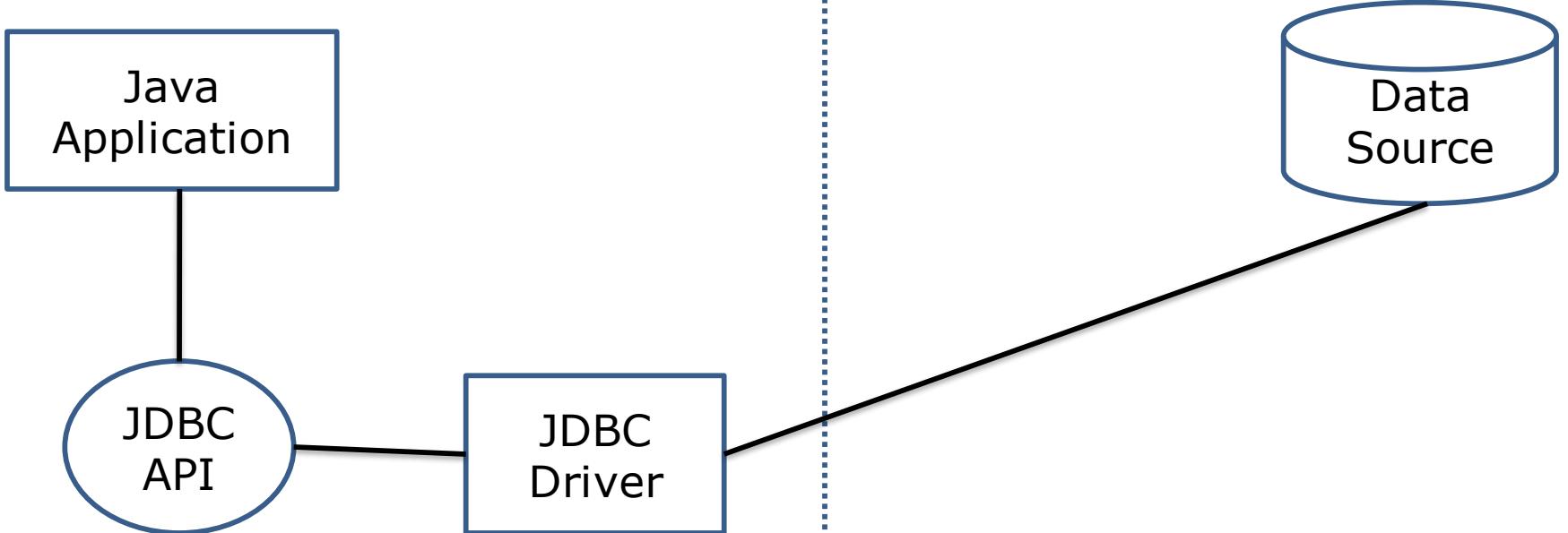
- **What does IDS Server do?**
- IDS Server is an internet database access server (or middleware) that allows you to create Web applications that can interact with your backend DBMS. IDS Server offers two ways of doing this:
  - (a) You can embed special tags in a normal HTML document which the IDS Server will recognize and process, then return database query results in a Web page that can be viewed by your browser. We call this technology IDS HTML Extensions.
  - (b) IDS Server includes IDS JDBC Driver, a Type-3 JDBC driver that will enable you to write and deploy Java applets and programs using the JDBC API to perform database operations.

## Type 4: 100% pure Java

- In a Type 4 driver, a pure Java-based driver that communicates directly with vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.
- MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.



**Figure Type 4: 100% pure Java**



**Figure Type 4: 100% pure Java**

## Which Driver should be used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

## **Connection:**

- The connection to the database is established by using one of three `getConnection()` methods of the `DriverManager` object. The `getConnection()` method requests access to the database from the DBMS. It is up to the DBMS to grant or reject access. A `Connection` object is returned by the `getConnection()` method if access is granted, otherwise the `getConnection()` method throws an `SQLException`.
- Sometimes the DBMS grants access to a database to anyone. In this case, the J2ME application uses the `getConnection(String url)` method. One parameter is passed to the method because the DBMS only needs the database identified. This is shown in Listing 10-5.
- `String url = "jdbc:odbc:CustomerInformation";`
- `Statement DataRequest;`
- `Connection Db;`
- `try {`
- `Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");`
- `Db = DriverManager.getConnection(url);`
- `}`
- `catch (ClassNotFoundException error) {`

- System.err.println("Unable to load the JDBC/ODBC bridge." +  
• error);
- System.exit(1);
- }
- catch (SQLException error) {
- System.err.println("Cannot connect to the database." + error);
- System.exit(2);
- }

- Sometimes the DBMS grants access to a database to anyone. In this case, the J2ME application uses the `getConnection(String url)` method. One parameter is passed to the method because the DBMS only needs the database identified. This is shown in Listing 10-5.
- ```
String url = "jdbc:odbc:CustomerInformation";
Statement DataRequest;
Connection Db;
try {
    Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
    Db = DriverManager.getConnection(url);
}
catch (ClassNotFoundException error) {
    System.err.println("Unable to load the JDBC/ODBC bridge." +
error);
    System.exit(1);
}
catch (SQLException error) {
    System.err.println("Cannot connect to the database." + error);
    System.exit(2);
}
```

- **TIME OUT**

- multiple applications might attempt to access a database simultaneously. The DBMS may not respond quickly for a number of reasons, one of which might be that database connections are not available. Rather than wait for a delayed response from the DBMS, the J2ME application can set a timeout period after which the Driver Manager will cease trying to connect to the database.
- The public static void DriverManager.setLoginTimeout(int seconds) method can be used by the J2ME application to establish the maximum time the DriverManager waits for a response from a DBMS before timing out. Likewise, the public static int DriverManager.getLoginTimeout() method is used to retrieve from the DriverManager the maximum time the DriverManager is set to wait until it times out. The DriverManager.getLoginTimeout() returns an int that represents seconds.

# Connection Pool

- A *connection pool* is a collection of database connections that are opened once and loaded into memory so these connections can be reused without having to reconnect to the DBMS.
- Clients use the Data Source interface to interact with the connection pool.
- The connection pool itself is implemented by the application server and other J2EE-specific technologies, which hide details on how the connection pool
- . There are two types of connections made to the database. The first is the physical connection, which is made by the application server using Pooled Connection objects.
- Pooled Connection objects are cached and reused. The other type of connection is the logical connection.

- Context ctext = new InitialContext();
- DataSource pool = (DataSource)  
ctext.lookup("java:comp/env/jdbc/pool");
- Connection db = pool.getConnection();
- // Place code to interact with the database  
here
- db.close();

- **Statement Objects**
- Once a connection to the database is opened, the J2ME application creates and sends a
- query to access data contained in the database. The query is written using SQL.
- One of three types of Statement objects is used to execute the query.
- These objects are Statement, which executes a query immediately;
- Prepared Statement, which is used to execute a compiled query; and Callable Statement, which is used to execute store procedures.

# The Statement Object

- The Statement object is used whenever a J2ME application needs to execute a query immediately without first having the query compiled. The Statement object contains the executeQuery() method, which is passed the query as an argument. The query is then transmitted to the DBMS for processing. The executeQuery() method returns one ResultSet object that contains rows, columns, and metadata that represent data requested by the query. The Result Set object also contains methods that are used to manipulate data in the ResultSet.
- The execute() method of the Statement object is used when multiple results may be returned. A third commonly used method of the Statement object is the executeUpdate() method. The executeUpdate() method is used to execute queries that contain UPDATE and DELETE SQL statements, which change values in a row and remove a row, respectively. The executeUpdate() method returns an integer indicating the number of rows that were updated by the query. ExecuteUpdate() is used to INSERT, UPDATE, DELETE, and DDL statements.

- String url = "jdbc:odbc:CustomerInformation";
- String userID = "jim";
- String password = "keogh";
- Statement DataRequest;
- ResultSet Results;
- Connection Db;
- try {
- Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
- Db = DriverManager.getConnection(url,userID,password);
- }
- catch (ClassNotFoundException error) {
- System.err.println("Unable to load the JDBC/ODBC bridge." +
- error);
- System.exit(1);
- }
- catch (SQLException error) {
- System.err.println("Cannot connect to the database." + error);

- System.exit(2);
- }
- try {
- String query = "SELECT \* FROM Customers";
- DataRequest = Db.createStatement();
- Results = DataRequest.executeQuery (query);
- //Place code here to interact with the ResultSet
- DataRequest.close();
- }
- catch ( SQLException error ){
- System.err.println("SQL error." + error);
- System.exit(3);
- }
- Db.close();

## Example for executeupdate() method

- String url = "jdbc:odbc:CustomerInformation";
- String userID = "jim";
- String password = "keogh";
- Statement DataRequest;
- Connection Db;
- int rowsUpdated;
- try {
- Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
- Db = DriverManager.getConnection(url,userID,password);
- }
- catch (ClassNotFoundException error) {
- System.err.println("Unable to load the JDBC/ODBC bridge." +
- error);
- System.exit(1);
- }

```
•    }
•    catch (SQLException error) {
•        System.err.println("Cannot connect to the database." + error);
•        System.exit(2);
•    }
•    try {
•        String query = "UPDATE Customers SET PAID='Y' WHERE BALANCE = '0';
•        DataRequest = Db.createStatement();
•        rowsUpdated = DataRequest.executeUpdate (query);
•        DataRequest.close();
•    }
•    catch ( SQLException error ){
•        System.err.println("SQL error." + error);
•        System.exit(3);
•    }
•    Db.close();
```

# **PreparedStatement Object**

- An SQL query must be compiled before the DBMS processes the query. Compiling occurs after one of the Statement object's execution methods is called. Compiling a query is an overhead that is acceptable if the query is called once. However, the compiling process can become an expensive overhead if the query is executed several times by the same instance of the J2ME application during the same session.
- An SQL query can be precompiled and executed by using the PreparedStatement object. The preparedStatement() method of the Connection object is called to return the PreparedStatement object. The preparedStatement() method is passed the query that is then precompiled. *The setXXX() method of the PreparedStatement object is used to replace the question mark with the value passed to the setXXX() method. There are a number of setXXX() methods available in the PreparedStatement object, each of which specifies the data type of the value that is being passed to the setXXX() method*

- String url = "jdbc:odbc:CustomerInformation";
- String userID = "jim";
- String password = "keogh";
- ResultSet Results;
- Connection Db;
- try {
- Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
- Db = DriverManager.getConnection(url(userID,password);
- }
- catch (ClassNotFoundException error) {
- System.out.println(
- "Unable to load the JDBC/ODBC bridge." + error);
- System.exit(1);
- }

```
catch (SQLException error) {  
    System.err.println("Cannot connect to the  
    database." + error);  
    System.exit(2);  
}  
try {  
    String query = "SELECT * FROM Customers WHERE  
    CustNumber = ?";  
    PreparedStatement pstatement =  
        Db.prepareStatement(query);  
    pstatement.setString(1, "123");  
    Results = pstatement.executeQuery();  
    //Place code here to interact with the ResultSet
```

- pstatement.close();
- }
- catch ( SQLException error ){
- System.err.println("SQL error." + error);
- System.exit(3);
- }
- Db.close();

# Callable Statement

- The Callable Statement is used to call a stored procedure from within a J2ME object. A stored procedure is a block of code and is identified by a unique name.
- The Callable Statement object uses three types of parameters when calling a stored procedure.
- These parameters are IN, OUT, and INOUT.

- The IN parameter contains any data that needs to be passed to the stored procedure and whose value is assigned using the *setXXX() method*.
- The OUT parameter contains the value returned by the stored procedures, if any. The OUT parameter must be registered using the *registerOutParameter()* method .
- J2ME application using the *getXXX() method*. *The INOUT parameter is a single parameter used for both passing information to the stored procedure and retrieving information from a stored procedure using the techniques described in the previous two paragraphs.*

# ResultSet

- A query is used to update, delete, and retrieve information stored in a database.
- The executeQuery() method is used to send the query to the DBMS for processing and returns a ResultSet object that
- contains data requested by the query.
- The ResultSet object contains methods that are used to copy data from the ResultSet into a Java collection of objects or variable(s) for further processing.
- Data in a ResultSet object is logically organized into a virtual table consisting of rows and columns.
- In addition to data, the ResultSet object also contains metadata, such as column names, column size, and column data type.

**The ResultSet uses a virtual cursor to point to a row of the virtual table.**

**A J2ME application must move the virtual cursor to each row, then use other methods of the ResultSet object to interact with the data stored in columns of that row.**

**The virtual cursor is positioned above the first row of data when the ResultSet is returned by the executeQuery() method.**

**This means that the virtual cursor must be moved to the first row using the next() method.**

**The next() method returns a boolean true if the row contains data, otherwise a boolean false is returned, indicating that no more rows exist in the ResultSet.**

**Columns appear in the ResultSet in the order in which column names appeared in the SELECT statement in the query.**

**Let's say a query contained the following SELECT statement:**

**SELECT CustomerFirstName, CustomerLastName FROM Customer**

# READING A RESULT SET

- Once a successful connection is made to the database, a query is defined in the second try {} block to retrieve the first name and last name of customers from the Customers table of the CustomerInformation database.
- The next() method of the ResultSet is called to move the virtual pointer to the first row in the ResultSet. If there is data in that row, the next() returns a true, which is assigned the boolean variable Records. If there isn't any data in that row, Records is assigned a false value.
- A false value is trapped by the if statement, where the “End of data.” message is displayed and the program terminates. A true value causes the program to enter the do...while in the third try {} block, where the getString() method is called to retrieve values in the first and second columns of the ResultSet. The values correspond to the first name and last name. These values are assigned to their corresponding String object, which is then concatenated and assigned the printrow String object and printed on the screen.

```
boolean Records = Results.next();
if (!Records ) {
    System.out.println("No data returned");
    System.exit(4);
}
try {
    do {
        FirstName = Results.getString ( 1 ) ;
        LastName = Results.getString ( 2 ) ;
        printrow = FirstName + " " + LastName;
        System.out.println(printrow);
    } while (Results.next() );
    DataRequest.close();
```

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
String printrow;
String FirstName;
String LastName;
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
    Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
    Db = DriverManager.getConnection(url(userID,password));
}
catch (ClassNotFoundException error) {
    System.err.println("Unable to load the JDBC/ODBC bridge." +
error);
    System.exit(1);
```

```
} catch (SQLException error) {
System.err.println("Cannot connect to the database." + error);
System.exit(2);
} try {
String query = "SELECT FirstName,LastName FROM Customers";
DataRequest = Db.createStatement();
Results = DataRequest.executeQuery (query);
}
catch ( SQLException error ){
System.err.println("SQL error." + error);
System.exit(3);
} boolean Records = Results.next();
if (!Records ) {
System.out.println("No data returned");
System.exit(4);
} try { do {
FirstName = Results.getString ( 1 ) ;
LastName = Results.getString ( 2 ) ;
printrow = FirstName + " " + LastName;
System.out.println(printrow);
} while (Results.next() );
DataRequest.close();
} catch (SQLException error ) {
System.err.println("Data display error." + error);
System.exit(5); }
```

## Scrollable ResultSet

Until the release of JDBC 2.1 API, the virtual cursor could only be moved down the ResultSet object.

But today the virtual cursor can be moved backwards or even positioned at a specific row.

The JDBC 2.1 API also enables a J2ME application to specify the number of rows to return from the DBMS.

Six methods of the ResultSet object are used to position the virtual cursor, in addition to the next() method discussed in the previous section.

These are first(), last(), previous(), absolute(), relative(), and getRow().

The first() method moves the virtual cursor to the first row in the ResultSet. Likewise, the last() method positions the virtual cursor at the last row in the ResultSet.

The previous() method moves the virtual cursor to the previous row.

The absolute() method positions the virtual cursor at the row number specified by the integer passed as a parameter to the absolute() method.

**The relative() method moves the virtual cursor the specified number of rows contained in the parameter.**

**The parameter is a positive or negative integer, where the sign represents the direction the virtual cursor is moved.**

**For example, a -4 moves the virtual cursor back four rows from the current row. Likewise, a 5 moves the virtual cursor forward five rows from the current row.**

**And the getRow() method returns an integer that represents the number of the current row in the ResultSet.**

**The Statement object that is created using the createStatement() of the Connection object must be set up to handle a scrollable ResultSet by passing the createStatement() method one of three constants.**

**These constants are TYPE\_FORWARD\_ONLY, TYPE\_SCROLL\_INSENSITIVE, and TYPE\_SCROLL\_SENSITIVE.**

The **TYPE\_FORWARD\_ONLY** constant restricts the virtual cursor to downward movement, which is the default setting. **TYPE\_SCROLL\_INSENSITIVE** and **TYPE\_SCROLL\_SENSITIVE** constants permit the virtual cursor to move in both directions. **TYPE\_SCROLL\_INSENSITIVE** makes the ResultSet insensitive to data changes made by another J2ME application in the table whose rows are reflected in the ResultSet. The **TYPE\_SCROLL\_SENSITIVE** constant makes the ResultSet sensitive to those changes.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
String printrow;
String FirstName;
String LastName;
Statement DataRequest;
```

```
ResultSet Results;
Connection Db;
try {
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
Db = DriverManager.getConnection(url,userID,password);
}
catch (ClassNotFoundException error) {
System.err.println("Unable to load the JDBC/ODBC bridge." +
error);
System.exit(1);
}
catch (SQLException error) {
System.err.println("Cannot connect to the database." + error);
System.exit(2);
}
try {
String query = "SELECT FirstName,LastName FROM Customers";
DataRequest = Db.createStatement(TYPE_SCROLL_INSENSITIVE);
Results = DataRequest.executeQuery (query);
}
catch ( SQLException error ){
System.err.println("SQL error." + error);
System.exit(3);
```

```
}

boolean Records = Results.next();
if (!Records ) {
System.out.println("No data returned");
System.exit(4);
}
try {
do {
Results.first();
Results.last();
Results.previous();
Results.absolute(10);
Results.relative(-2);
Results.relative(2);
FirstName = Results.getString ( 1 ) ;
LastName = Results.getString ( 2 ) ;
printrow = FirstName + " " + LastName;
System.out.println(printrow);
} while (Results.next() );
DataRequest.close();
}
catch (SQLException error ) {
System.err.println("Data display error." + error);
System.exit(5);
}
```

## Not All JDBC Drivers Are Scrollable

Although the JDBC API contains methods to scroll a ResultSet, some JDBC drivers may not support some or all of these features, and therefore they will not be able to return a scrollable ResultSet.

Listing 10-15 can be used to test whether or not the JDBC driver in use supports a scrollable ResultSet.

```
boolean forward, insensitive, sensitive;
```

```
DataBaseMetaData meta = Db.getMetaData();
```

```
forward =
```

```
meta.supportsResultsSetType(ResultSet.TYPE_FORWARD_ONLY);
```

```
insensitive = meta.supportsResultsSetType(
```

```
ResultSet.TYPE_SCROLL_INSENSITIVE);
```

```
sensitive = meta.supportsResultsSetType(
```

```
ResultSet.TYPE_SCROLL_SENSITIVE);
```

```
System.out.println("forward: " + answer);
```

```
System.out.println("insensitive: " + insensitive);
```

```
System.out.println("sensitive: " + sensitive);
```

# Updatable ResultSet

- Rows contained in the ResultSet can be updated similar to how rows in a table can be updated. This is made possible by passing the createStatement() method of the Connection object the CONCUR\_UPDATABLE. Alternatively, the CONCUR\_READ\_ONLY constant can be passed to the createStatement() method to prevent the ResultSet from being updated.
- There are three ways to update a ResultSet. These are updating values in a row, deleting a row, and inserting a new row. All of these changes are accomplished by using methods of the Statement object.
- The updateRow() method is called after all the updateXXX() *methods are called*. The updateRow() method changes values in columns of the current row of the ResultSet based on the values of the updateXXX() *methods*.
- The updateString() method is used to change the value of the last name column of the ResultSet to ‘Smith’. The change takes effect once the updateRow() method is called, but this change only occurs in the ResultSet

```
• String url = "jdbc:odbc:CustomerInformation";
• String userID = "jim";
• String password = "keogh";
• String printrow;
• String FirstName;
• String LastName;
• Statement DataRequest;
• ResultSet Results;
• Connection Db;
• try {
•   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
•   Db = DriverManager.getConnection(url(userID,password);
• }
• catch (ClassNotFoundException error) {
•   System.err.println("Unable to load the JDBC/ODBC bridge." +
•   error);
•   System.exit(1);
• }
```

- catch (SQLException error) {
- System.out.println("Cannot connect to the database." + error);
- System.exit(2);
- }
- try {
- String query = "SELECT FirstName,LastName FROM Customers";
- DataRequest = Db.createStatement();
- Results = DataRequest.executeQuery (query);
- }
- catch ( SQLException error ){
- System.out.println("SQL error." + error);
- System.exit(3);
- }
- boolean Records = Results.next();
- if (!Records ) {
- System.out.println("No data returned");
- System.exit(4);
- }

- try {
- do {
- FirstName = Results.getString ( 1 ) ;
- LastName = Results.getString ( 2 ) ;
- printrow = FirstName + " " + LastName;
- System.out.println(printrow);
- } while (Results.next() );
- DataRequest.close();
- }
- catch (SQLException error ) {
- System.err.println("Data display error." + error);
- System.exit(5);
- }

# Delete a Row in the ResultSet

- The **deleteRow()** method is used to remove a row from a ResultSet. Sometimes this is advantageous when processing the ResultSet because this is a way to eliminate rows from future processing.
- The **deleteRow()** method is passed an integer that contains the number of the row to be deleted. The **deleteRow()** method is passed an integer that contains the number of the row to be deleted.
- 
- **Results.deleteRow(0);**

# Insert a Row in the ResultSet

- Inserting a row into the ResultSet is accomplished using basically the same technique used to update the ResultSet. That is, the *updateXXX()* method is used to specify the column and value that will be placed into the column of the ResultSet.
- The *updateXXX()* method requires two parameters.
- *The first parameter is either the name of the column or the number of the column of the ResultSet.*
- The second parameter is the new value that will be placed in the column of the ResultSet..

- String url = "jdbc:odbc:CustomerInformation";
- String userID = "jim";
- String password = "keogh";
- Statement DataRequest;
- ResultSet Results;
- Connection Db;
- try {
- Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
- Db = DriverManager.getConnection(url,userID,password);
- }
- catch (ClassNotFoundException error) {
- System.err.println("Unable to load the JDBC/ODBC bridge." +
- error);
- System.exit(1);
- }

```
• catch (SQLException error) {  
•     System.err.println("Cannot connect to the database." + error);  
•     System.exit(2);  
• }  
• try {  
•     String query = "SELECT FirstName,LastName FROM Customers";  
•     DataRequest = Db.createStatement(CONCUR_UPDATABLE);  
•     Results = DataRequest.executeQuery (query);  
• }  
• catch ( SQLException error ){  
•     System.err.println("SQL error." + error);  
•     System.exit(3);  
• }  
• boolean Records = Results.next();  
• if (!Records ) {  
•     System.out.println("No data returned");
```

- String url = "jdbc:odbc:CustomerInformation";
- String userID = "jim";
- String password = "keogh";
- Statement DataRequest;
- ResultSet Results;
- Connection Db;
- try {
- Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
- Db = DriverManager.getConnection(url(userID,password);
- }
- catch (ClassNotFoundException error) {
- System.err.println("Unable to load the JDBC/ODBC bridge." +
- error);
- System.exit(1);
- );

```
• }
• catch (SQLException error) {
•     System.err.println("Cannot connect to the database." + error);
•     System.exit(2
• }
• try {
•     String query = "SELECT FirstName,LastName FROM Customers";
•     DataRequest = Db.createStatement(CONCUR_UPDATABLE);
•     Results = DataRequest.executeQuery (query);
• }
• catch ( SQLException error ){
•     System.err.println("SQL error." + error);
•     System.exit(3);
```

- }
- boolean Records = Results.next();
- if (!Records ) {
- System.out.println("No data returned");
- System.exit(4);
- }
- try {
- Results.updateString (1, "Tom");
- Results.updateString (2, "Smith");
- Results.insertRow();
- DataRequest.close();
- }
- catch (SQLException error ) {

- System.err.println("Data display error." + error);
- System.exit(5);
- }
- System.exit(4);
- }
- try {
- Results.updateString (1, "Tom");
- Results.updateString (2, "Smith");
- Results.insertRow();
- DataRequest.close();
- }
- catch (SQLException error ) {
- System.err.println("Data display error." + error);
- System.exit(5);
- }

# **JSP-JAVA SERVER PAGE**

JSP: Understanding Java Server Pages-  
JSP Standard Tag Library(JSTL)-Creating  
HTML forms by embedding JSP code.

- **Java Server Page (JSP)** is a server-side technology.
- **Java Server Pages** are an extension to the Java servlet technology that was developed by Sun.
- JavaServer Pages (**JSP**) is a technology that helps software developers create dynamically generated web pages based on HTML, XML, or other document types. Released in 1999 by Sun Microsystems.
- **JSP** is similar to PHP and ASP, but it uses the Java programming language.

- A JSP page consists of HTML tags and JSP tags. The jsp pages are easier to maintain than servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tag etc.

## **Advantage of JSP over Servlet**

- There are many advantages of JSP over servlet. They are as follows:

### **1) Extension to Servlet**

- We can use all the features of servlet in JSP. In addition to, we can use implicit objects, predefined tags, expression language and Custom tags in JSP, that makes JSP development easy.

### **2) Easy to maintain**

- JSP can be easily managed because we can easily separate our business logic with presentation logic. In servlet technology, we mix our business logic with the presentation logic.

### 3) Fast Development: No need to recompile and redeploy

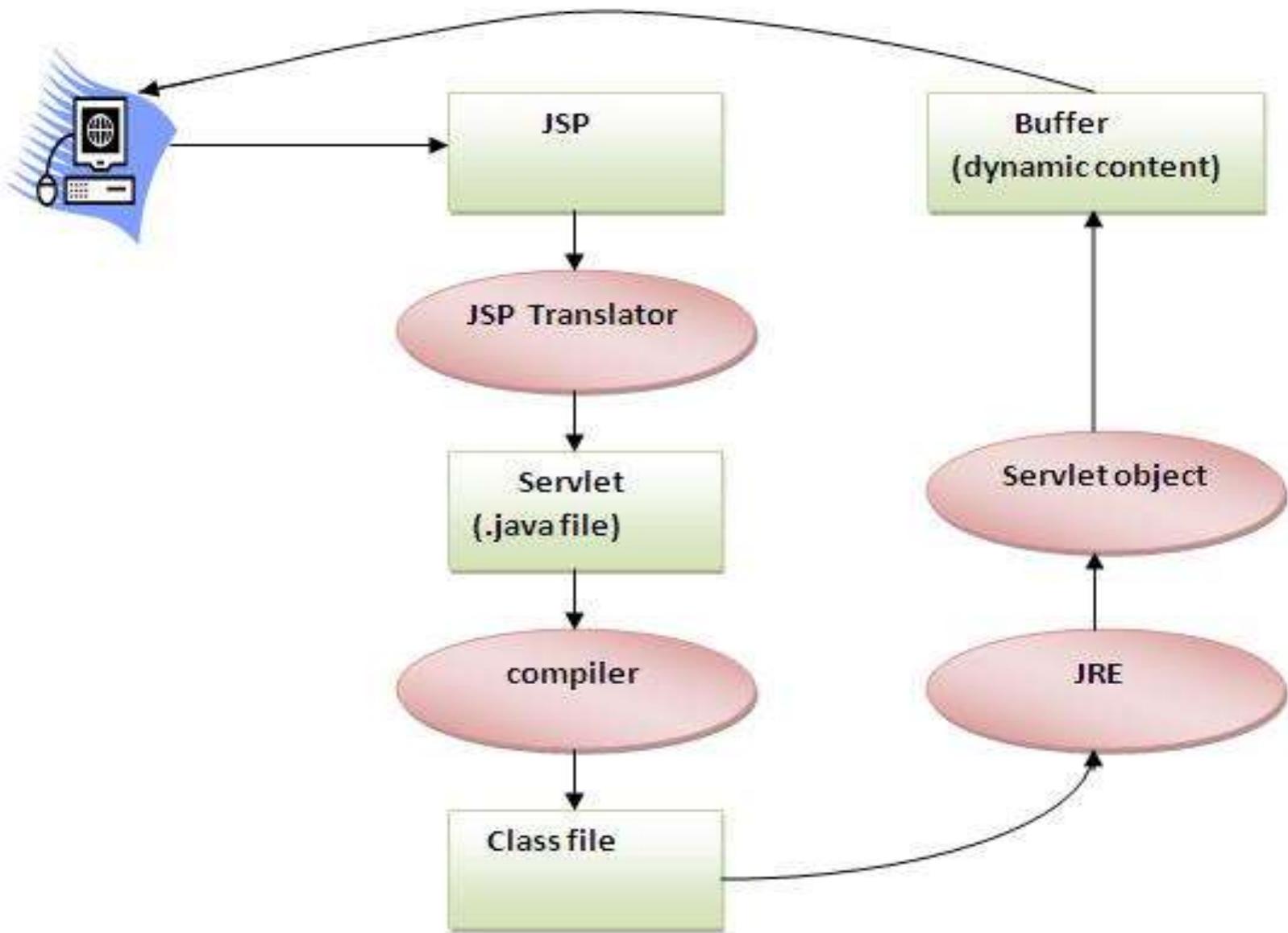
- If JSP page is modified, we don't need to recompile and redeploy the project. The servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

### 4) Less code than Servlet

- In JSP, we can use a lot of tags such as action tags, jstl, custom tags etc. that reduces the code. Moreover, we can use EL, implicit objects etc.

# Life cycle of a JSP Page

- The JSP pages follows these phases:
- Translation of JSP Page
- Compilation of JSP Page
- Classloading (class file is loaded by the classloader)
- Instantiation (Object of the Generated Servlet is created).
- Initialization ( `jsplInit()` method is invoked by the container).
- Request processing ( `_jspService()` method is invoked by the container).
- Destroy ( `jspDestroy()` method is invoked by the container).



- As depicted in the above diagram, JSP page is translated into servlet by the help of JSP translator.
- The JSP translator is a part of webserver that is responsible to translate the JSP page into servlet.
- After that Servlet page is compiled by the compiler and gets converted into the class file.
- Moreover, all the processes that happens in servlet is performed on JSP later like initialization, committing response to the browser and destroy.

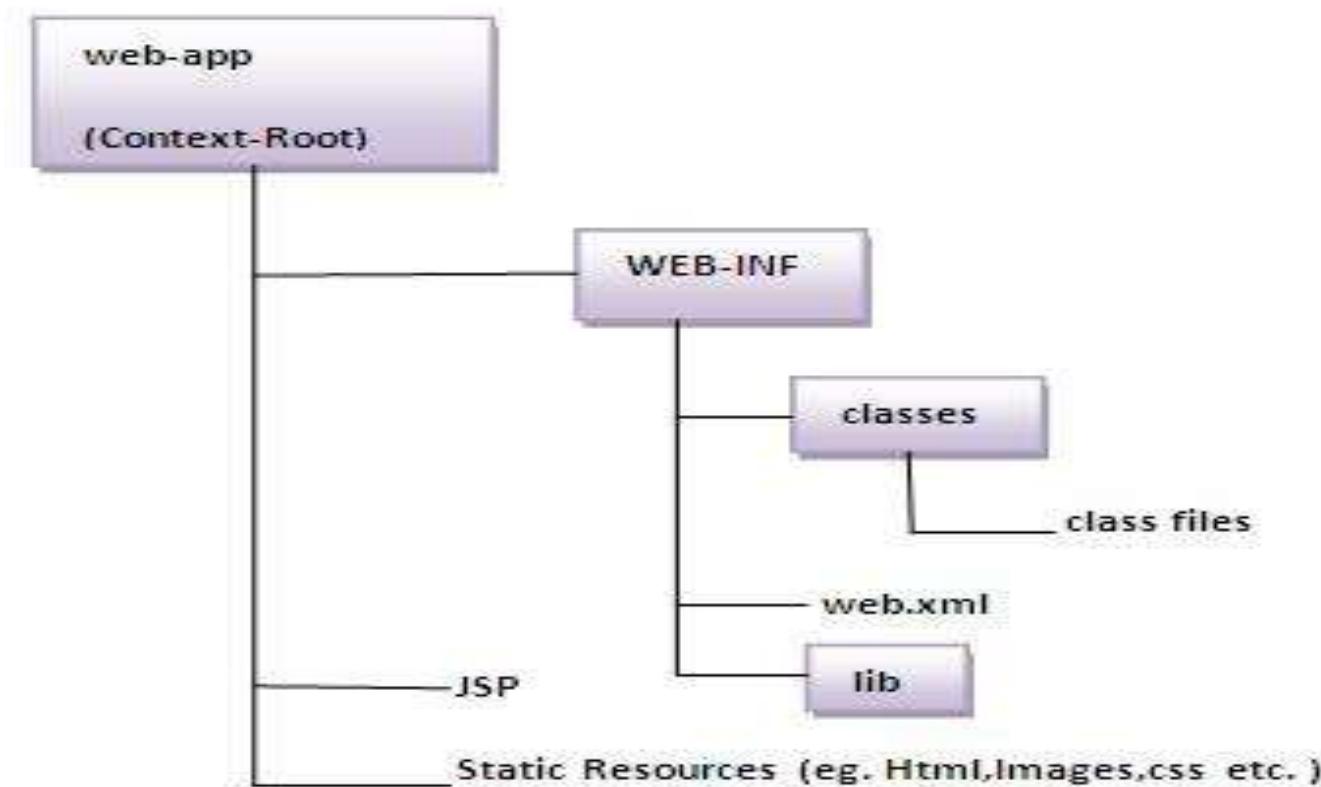
- **Creating a simple JSP Page**
- To create the first jsp page, write some html code as given below, and save it by .jsp extension. We have save this file as index.jsp. Put it in a folder and paste the folder in the web-apps directory in apache tomcat to run the jsp page.
- **index.jsp** Let's see the simple example of JSP, here we are using the scriptlet tag to put java code in the JSP page. We will learn scriptlet tag later.
  - <html>
  - <body>
  - <% out.print(2\*5); %>
  - </body>
  - </html>
- It will print **10** on the browser.

# How to run a simple JSP Page ?

- Follow the following steps to execute this JSP page:
- Start the server
- put the jsp file in a folder and deploy on the server
- visit the browser by the url  
`http://localhost:portno/contextRoot/jspfile` e.g.  
`http://localhost:8888/myapplication/index.jsp`

# Directory structure of JSP

- The directory structure of JSP page is same as servlet. We contains the jsp page outside the WEB-INF folder or in any directory.



# The JSP API

- The JSP API consists of two packages:

1. `javax.servlet.jsp`
2. `javax.servlet.jsp.tagext`
3. `javax.servlet.jsp.el`

javax.servlet.jsp package

- The javax.servlet.jsp package has two interfaces and classes.
- The two interfaces are as follows:

JspPage  
HttpJspPage

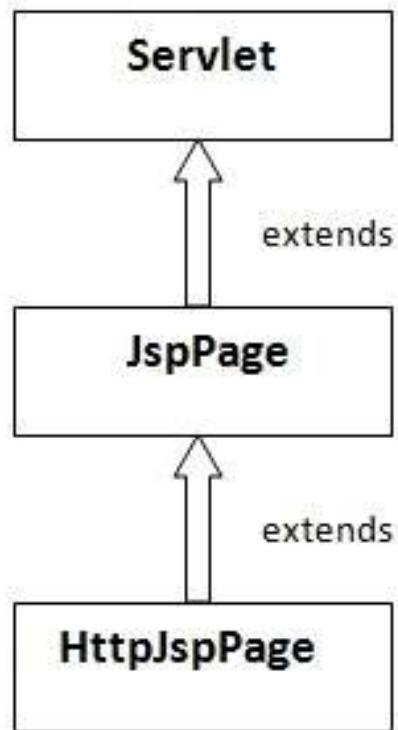
- The classes are as follows:

JspWriter  
PageContext  
JspFactory  
JspEngineInfo  
JspException  
JspError

- Exception classes:jsp,jsptag,skipage exception classes

# The JspPage interface

- According to the JSP specification, all the generated servlet classes must implement the JspPage interface. It extends the Servlet interface. It provides two life cycle methods.



# Methods of JspPage interface

- **public void jspInit():** It is invoked only once during the life cycle of the JSP when JSP page is requested firstly. It is used to perform initialization. It is same as the init() method of Servlet interface.
- **public void jspDestroy():** It is invoked only once during the life cycle of the JSP before the JSP page is destroyed. It can be used to perform some clean up operation.

# The HttpJspPage interface

- The HttpJspPage interface provides the one life cycle method of JSP. It extends the JspPage interface.
- Method of HttpJspPage interface:
- **public void \_jspService():** It is invoked each time when request for the JSP page comes to the container. It is used to process the request. The underscore \_ signifies that you cannot override this method.

# JSP Scriptlet tag (Scripting elements)

- In JSP, java code can be written inside the jsp page using the scriptlet tag. Let's see what are the scripting elements first.
- **JSP Scripting elements**
- The scripting elements provides the ability to insert java code inside the jsp. There are three types of scripting elements:
  - **scriptlet tag**
  - **expression tag**
  - **declaration tag**

- JSP scriptlet tag
- A scriptlet tag is used to execute java source code in JSP. Syntax is as follows:
- <% java source code %>
- Example of JSP scriptlet tag
- In this example, we are displaying a welcome message.
- <html>
- <body>
- <% out.print("welcome to jsp"); %>
- </body>
- </html>

Example of JSP scriptlet tag that prints the user name

*File: index.html*

**<html>**

**<body>**

**<form action="welcome.jsp">**

**<input type="text" name="uname">**

**<input type="submit" value="go"><br/>**

**</form>**

**</body>**

**</html>**

*File: welcome.jsp*

```
<html>
<body>
<%
String name=request.getParameter("uname");
out.print("welcome "+name);
%>
</form>
</body>
</html>
```

# JSP expression tag

The code placed within **JSP expression tag** is *written to the output stream of the response*. So you need not write out.print() to write data. It is mainly used to print the values of variable or method.

## Syntax of JSP expression tag

<%= statement %>

### Example of JSP expression tag

In this example of jsp expression tag, we are simply displaying a welcome message.

```
<html>
<body>
<%= "welcome to jsp" %>
</body>
</html>
```

# Example of JSP expression tag that prints current time

To display the current time, we have used the `getTime()` method of `Calendar` class. The `getTime()` is an instance method of `Calendar` class, so we have called it after getting the instance of `Calendar` class by the `getInstance()` method.

*index.jsp*

**<html>**

**<body>**

Current Time: `<%= java.util.Calendar.getInstance().getTime() %>`

**</body>**

**</html>**

# JSP Declaration Tag

The **JSP declaration tag** is used *to declare fields and methods.*

The code written inside the jsp declaration tag is placed outside the service() method of auto generated servlet.

So it doesn't get memory at each request.

## Syntax of JSP declaration tag

The syntax of the declaration tag is as follows:

**<%! field or method declaration %>**

# Difference between JSP Scriptlet tag and Declaration tag

Jsp Scriptlet Tag	Jsp Declaration Tag
The jsp scriptlet tag can only declare variables not methods.	The jsp declaration tag can declare variables as well as methods.
The declaration of scriptlet tag is placed inside the <code>_jspService()</code> method.	The declaration of jsp declaration tag is placed outside the <code>_jspService()</code> method.

# Example of JSP declaration tag that declares field

In this example of JSP declaration tag, we are declaring the field and printing the value of the declared field using the jsp expression tag.

**index.jsp**

**<html>**

**<body>**

**<%! int data=50; %>**

**<%= "Value of the variable is:"+data %>**

**</body>**

**</html>**



# Topics

- JSP Fundamentals
- JSP Scripting Elements
- JSP Implicit Objects
- JSP Directives
- JSP Actions
- JSP Example (Loan Calculator)
- Servlets & JSPs together
- Tag Libraries
- Deploying and Running a JSP Application

# Java Server Pages (JSP)

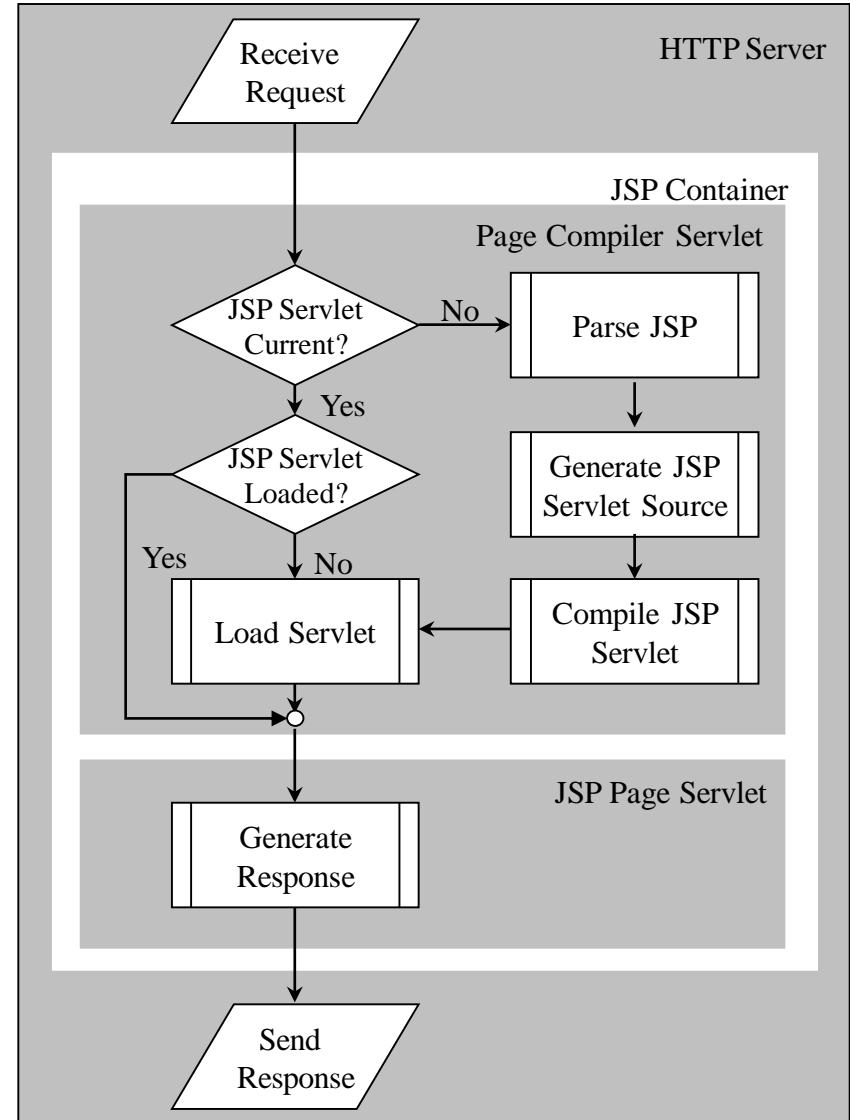
## Fundamentals

- Java Server Pages are HTML pages embedded with snippets of Java code.
  - It is an inverse of a Java Servlet
- Four different elements are used in constructing JSPs
  - Scripting Elements
  - Implicit Objects
  - Directives
  - Actions

# Java Server Pages (JSP)

## Architecture

- JSPs run in two phases
  - Translation Phase
  - Execution Phase
- In translation phase JSP page is compiled into a servlet
  - called JSP Page Implementation class
- In execution phase the compiled JSP is processed



# Scripting Elements

## Types

- There are three kinds of scripting elements
  - Declarations
  - Scriptlets
  - Expressions

# Declarations

## Basics

- Declarations are used to define methods & instance variables
  - Do not produce any output that is sent to client
  - Embedded in <%! and %> delimiters

Example:

```
<%!
    Public void jspDestroy() {
        System.out.println("JSP Destroyed");
    }
    Public void jspInit() {
        System.out.println("JSP Loaded");
    }
    int myVar = 123;
%>
```

- The functions and variables defined are available to the JSP Page as well as to the servlet in which it is compiled

# Scriptlets

## Basics

- Used to embed java code in JSP pages.
  - Contents of JSP go into `_JSPpageservice()` method
  - Code should comply with syntactical and semantic construct of java
  - Embedded in `<%` and `%>` delimiters

Example:

```
<%
    int x = 5;
    int y = 7;
    int z = x + y;
%>
```

# Expressions

## Basics

- Used to write dynamic content back to the browser.
  - If the output of expression is Java primitive the value is printed back to the browser
  - If the output is an object then the result of calling `toString` on the object is output to the browser
  - Embedded in `<%=` and `%>` delimiters

Example:

- `<%="Fred"+ " " + "Flintstone %>`  
prints “Fred Flintstone” to the browser
- `<%=Math.sqrt(100)%>`  
prints 10 to the browser

# Java Implicit Objects

## Scope

- Implicit objects provide access to server side objects
  - e.g. request, response, session etc.
- There are four scopes of the objects
  - Page: Objects can only be accessed in the page where they are referenced
  - Request: Objects can be accessed within all pages that serve the current request.  
(Including the pages that are forwarded to and included in the original jsp page)
  - Session: Objects can be accessed within the JSP pages for which the objects are defined
  - Application: Objects can be accessed by all JSP pages in a given context

# Java Implicit Objects

## List

- request: Reference to the current request
- response: Response to the request
- session: session associated woth current request
- application: Servlet context to which a page belongs
- pageContext: Object to access request, response, session and application associated with a page
- config: Servlet configuration for the page
- out: Object that writes to the response output stream
- page: instance of the page implementation class (this)
- exception: Available with JSP pages which are error pages

# Java Implicit Objects

## Example

```
<html>
  <head>
    <title>Implicit Objects</title>
  </head>
  <body style="font-family:verdana;font-size:10pt">
    <p>
      Using Request parameters...<br>
      <b>Name:</b> <%= request.getParameter("name") %>
    </p>
    <p>
      <% out.println("This is printed using the out implicit
        variable"); %>
    </p>
    <p>
      Storing a string to the session...<br>
      <% session.setAttribute("name", "Meeraj"); %>
    </p>
    <p>
      Retrieving the string from session...<br>
      <b>Name:</b> <%= session.getAttribute("name") %>
    </p>
    <p>
      Storing a string to the application...<br>
      <% application.setAttribute("name", "Meeraj"); %>
    </p>
    <p>
      Retrieving the string from application...<br>
      <b>Name:</b>
      <%= application.getAttribute("name") %>
    </p>
    <p>
      Storing a string to the page context...<br>
      <% pageContext.setAttribute("name", "Meeraj"); %>
    </p>
    <p>
      Retrieving the string from page context...<br>
      <b>Name:</b>
      <%= pageContext.getAttribute("name") %>
    </p>
  </body>
</html>
```

# Example Implicit Objects

## Deploy & Run

- Save file:
  - \$TOMCAT\_HOME/webapps/jsp/Implicit.jsp
- Access file
  - http://localhost:8080/jsp/Implicit.jsp?name=Sanjay
- Results of the execution

Using Request parameters...

**Name:** sanjay

This is printed using the out implicit variable

Storing a string to the session...

Retrieving the string from session...

**Name:** Meeraj

Storing a string to the application...

Retrieving the string from application...

**Name:** Meeraj

Storing a string to the page context...

Retrieving the string from page context...

**Name:** Meeraj

# Directives

## Basics & Types

- Messages sent to the JSP container
  - Aids the container in page translation
- Used for
  - Importing tag libraries
  - Import required classes
  - Set output buffering options
  - Include content from external files
- The jsp specification defines three directives
  - Page: provider information about page, such as scripting language that is used, content type, or buffer size
  - Include – used to include the content of external files
  - Taglib – used to import custom actions defined in tag libraries

# Page Directives

## Basics & Types

- Page directive sets page properties used during translation
  - JSP Page can have any number of directives
  - Import directive can only occur once
  - Embedded in <%@ and %> delimiters
- Different directives are
  - Language: (Default Java) Defines server side scripting language (e.g. java)
  - Extends: Declares the class which the servlet compiled from JSP needs to extend
  - Import: Declares the packages and classes that need to be imported for using in the java code (comma separated list)
  - Session: (Default true) Boolean which says if the session implicit variable is allowed or not
  - Buffer: defines buffer size of the jsp in kilobytes (if set to none no buffering is done)

# Page Directives

Types con't.

- Different directives are (cont'd.)
  - autoFlush: When true the buffer is flushed when max buffer size is reached (if set to false an exception is thrown when buffer exceeds the limit)
  - isThreadSafe: (default true) If false the compiled servlet implements SingleThreadModel interface
  - Info: String returned by the getServletInfo() of the compiled servlet
  - errorPage: Defines the relative URI of web resource to which the response should be forwarded in case of an exception
  - contentType: (Default text/html) Defines MIME type for the output response
  - isErrorPage: True for JSP pages that are defined as error pages
  - pageEncoding: Defines the character encoding for the jsp page

# Page Directives

## Example

```
<%@  
    page language="java"  
    buffer="10kb"  
    autoflush="true"  
    errorPage="/error.jsp"  
    import="java.util.*, javax.sql.RowSet"  
%>
```

# Include Directive

## Basics

- Used to insert template text and JSP code during the translation phase.
  - The content of the included file specified by the directive is included in the including JSP page
- Example
  - `<%@ include file="included.jsp" %>`

# JSP Actions

## Basics & Types

- Processed during the request processing phase.
  - As opposed to JSP directives which are processed during translation
- Standard actions should be supported by J2EE compliant web servers
- Custom actions can be created using tag libraries
- The different actions are
  - Include action
  - Forward action
  - Param action
  - useBean action
  - getProperty action
  - setProperty action
  - plugIn action

# JSP Actions

## Include

- Include action used for including resources in a JSP page
  - Include directive includes resources in a JSP page at translation time
  - Include action includes response of a resource into the response of the JSP page
  - Same as including resources using RequestDispatcher interface
  - Changes in the included resource reflected while accessing the page.
  - Normally used for including dynamic resources
- Example
  - <jsp:include page="inlcudedPage.jsp">
  - Includes the the output of includedPage.jsp into the page where this is included.

# JSP Actions

## Forward

- Forwards the response to other web specification resources
  - Same as forwarding to resources using RequestDispatcher interface
- Forwarded only when content is not committed to other web application resources
  - Otherwise an IllegalStateException is thrown
  - Can be avoided by setting a high buffer size for the forwarding jsp page
- Example
  - <jsp:forward page="Forwarded.html">
  - Forwards the request to Forwarded.html

# JSP Actions

## Param

- Used in conjunction with Include & Forward actions to include additional request parameters to the included or forwarded resource
- Example

```
<jsp:forward page="Param2.jsp">  
    <jsp:param name="FirstName" value="Sanjay">  
</jsp:forward>
```

- This will result in the forwarded resource having an additional parameter FirstName with a value of Sanjay

# JSP Actions

## useBean

- Creates or finds a Java object with the defined scope.
  - Object is also available in the current JSP as a scripting variable
- Syntax:

```
<jsp:useBean id="name"  
scope="page | request | session | application"  
class="className" type="typeName" |  
bean="beanName" type="typeName" |  
type="typeName" />
```

- At least one of the type and class attributes must be present
- We can't specify values for both the class and bean name.
- Example

```
<jsp:useBean id="myName" scope="request" class="java.lang.String">  
  <% firstName="Sanjay"; %>  
</jsp:useBean>
```

# JSP Actions

## get/setProperty

- getProperty is used in conjunction with useBean to get property values of the bean defined by the useBean action
  - Example (getProperty)
    - <jsp:getProperty name="myBean" property="firstName" />
    - Name corresponds to the id value in the useBean
    - Property refers to the name of the bean property
- setProperty is used to set bean properties
  - Example (setProperty)
    - <jsp:setProperty name="myBean" property="firstName" value="Sanjay"/>
    - Sets the name property of myBean to Sanjay
    - <jsp:setProperty name="myBean" property="firstName" param="fname"/>
    - Sets the name property of myBean to the request parameter fname
    - <jsp:setProperty name="myBean" property="\*"/>
    - Sets property to the corresponding value in request

# JSP Actions

## plugIn

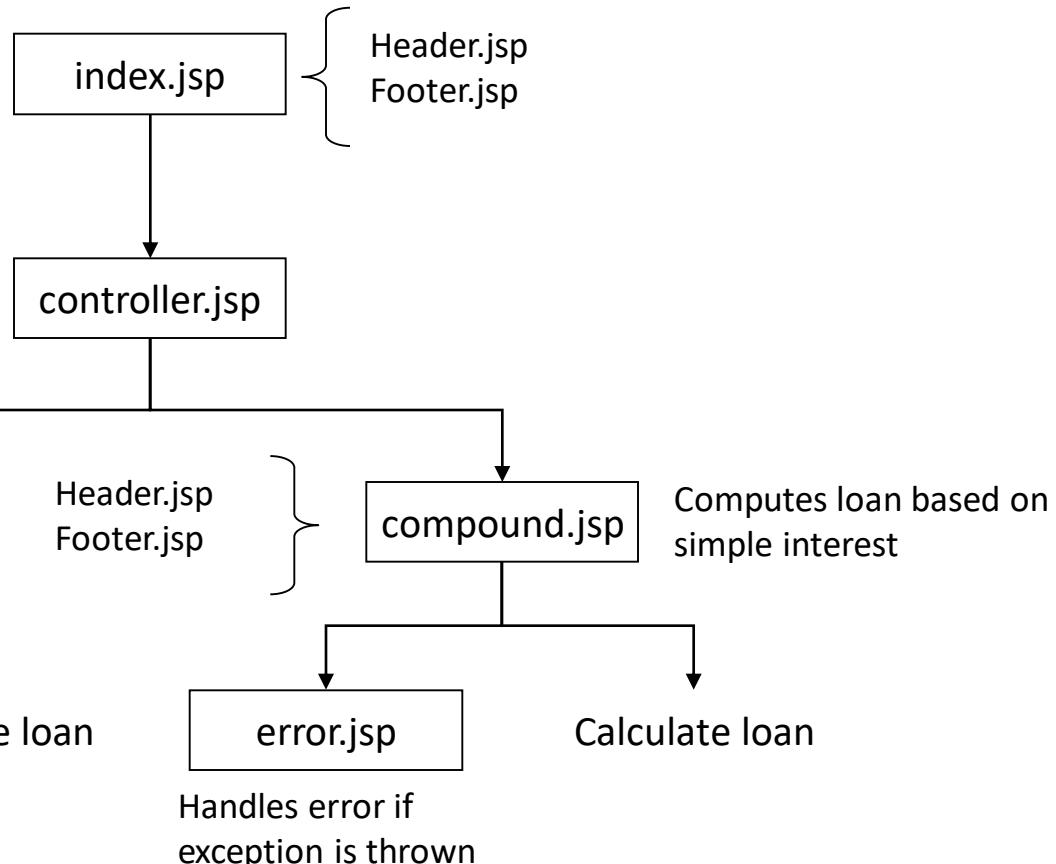
- Enables the JSP container to render appropriate HTML (based on the browser type) to:
  - Initiate the download of the Java plugin
  - Execution of the specified applet or bean
- plugIn standard action allows the applet to be embedded in a browser neutral fashion
- Example

```
<jsp:plugin type="applet" code="MyApplet.class" codebase="/">  
    <jsp:params>  
        <jsp:param name="myParam" value="122"/>  
    </jsp:params>  
    <jsp:fallback><b>Unable to load applet</b></jsp:fallback>  
</jsp:plugin>
```

# Example

## Loan Calculator

Gets input to compute loan from user →



Selects the right jsp for calculating loan →

Computes loan based on simple interest

Computes loan based on simple interest

# Loan Calculator

## index.jsp

```
<html>
  <head>
    <title>Include</title>
  </head>
  <body style="font-family:verdana;font-size:10pt;">
    <%@ include file="header.html" %>
    <form action="controller.jsp">
      <table border="0" style="font-family:verdana;font-size:10pt;">
        <tr>
          <td>Amount:</td>
          <td><input type="text" name="amount" /></td>
        </tr>
        <tr>
          <td>Interest in %:</td>
          <td><input type="text" name="interest"/></td>
        </tr>
        <tr>
          <td>Compound:</td>
          <td><input type="radio" name="type" value="C" checked/></td>
        </tr>
      </table>
      <tr>
        <td>Simple:</td>
        <td><input type="radio" name="type" value="S" /></td>
      </tr>
      <tr>
        <td>Period:</td>
        <td><input type="text" name="period"/></td>
      </tr>
      </table>
      <input type="submit" value="Calculate"/>
    </form>
    <jsp:include page="footer.jsp"/>
  </body>
</html>
```

# Loan Calculator

## Miscelaneous

### controller.jsp

```
<%
String type = request.getParameter("type");
if(type.equals("S")) {
%>
<jsp:forward page="/simple.jsp"/>
<%
} else {
%>
<jsp:forward page="/compound.jsp"/>
<%
}
%>
```

### error.jsp

```
<%@ page isErrorPage="true" %>
<html>
<head>
<title>Simple</title>
</head>
<body style="font-family:verdana;font-size:10pt;">
<%@ include file="header.html" %>
<p style="color=#FF0000"><b><%=
exception.getMessage() %></b></p>
<jsp:include page="footer.jsp"/>
</body>
</html>
header.jsp
<h3>Loan Calculator</h3>
footer.jsp
<%= new java.util.Date() %>
```

# Loan Calculator

## simple.jsp

```
<%@ page errorPage="error.jsp" %>
<%!
public double calculate(double amount, double
    interest, int period) {
    if(amount <= 0) {
        throw new IllegalArgumentException("Amount
            should be greater than 0: " + amount);
    }
    if(interest <= 0) {
        throw new IllegalArgumentException("Interest
            should be greater than 0: " + interest);
    }
    if(period <= 0) {
        throw new IllegalArgumentException("Period should
            be greater than 0: " + period);
    }
    return amount*(1 + period*interest/100);
%>
```

```
<html>
    <head>
        <title>Simple</title>
    </head>
    <body style="font-family:verdana;font-size:10pt;">
        <%@ include file="header.html" %>
        <%
            double amount =
                Double.parseDouble(request.getParameter("amount"));
            double interest =
                Double.parseDouble(request.getParameter("interest"));
            int period = Integer.parseInt(request.getParameter("period"));
        %>
        <b>Pincipal using simple interest:</b>
        <%= calculate(amount, interest, period) %>
        <br/><br/>
        <jsp:include page="footer.jsp"/>
    </body>
</html>
```

# Loan Calculator

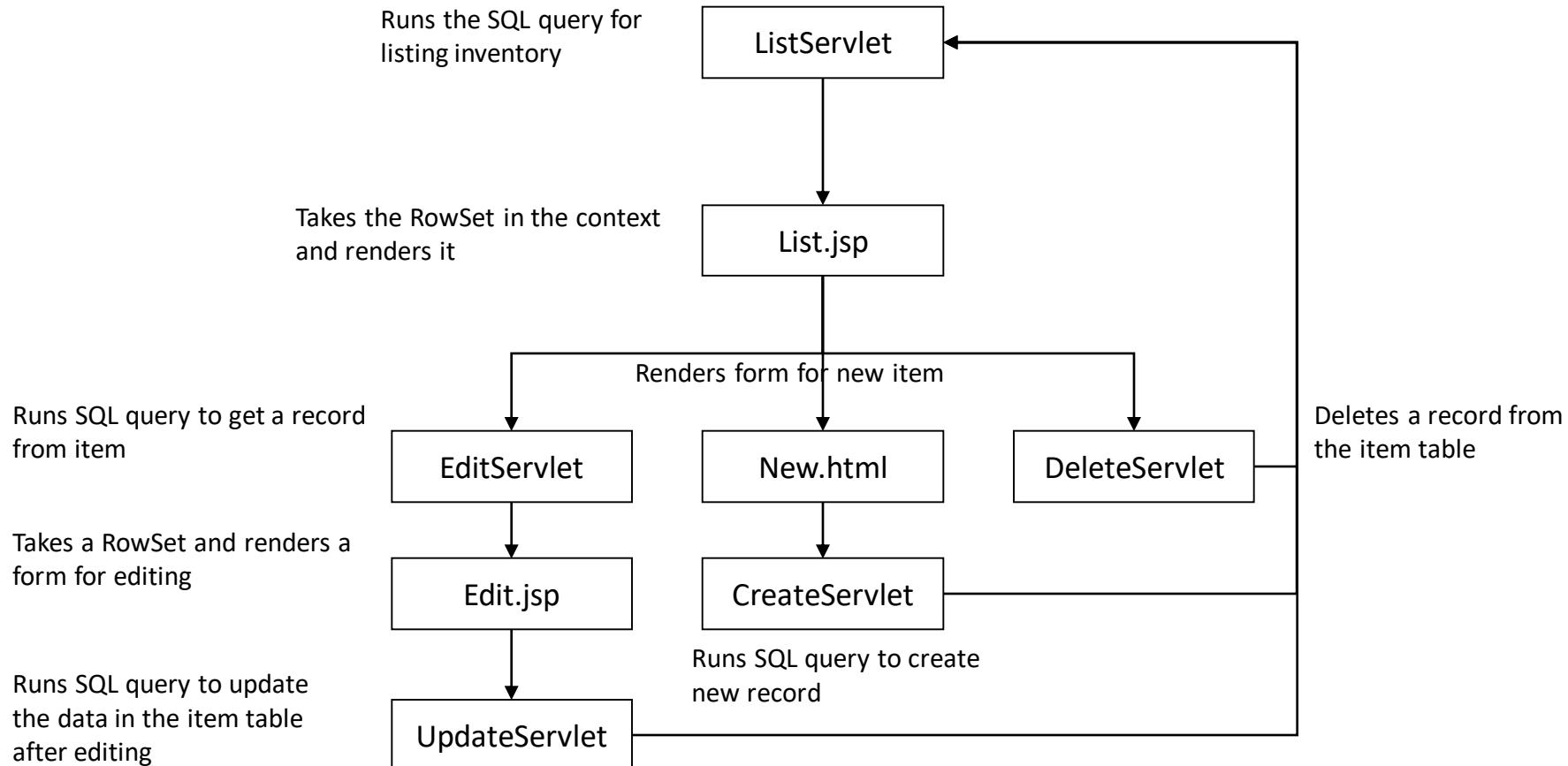
## compound.jsp

```
<%@ page errorPage="error.jsp" %>
<%!
public double calculate(double amount, double
    interest, int period) {
    if(amount <= 0) {
        throw new IllegalArgumentException("Amount
            should be greater than 0: " + amount);
    }
    if(interest <= 0) {
        throw new IllegalArgumentException("Interest
            should be greater than 0: " + interest);
    }
    if(period <= 0) {
        throw new IllegalArgumentException("Period should
            be greater than 0: " + period);
    }
    return amount*Math.pow(1 + interest/100, period);
%>
```

```
<html>
    <head>
        <title>Compound</title>
    </head>
    <body style="font-family:verdana;font-size:10pt;">
        <%@ include file="header.html" %>
        <%
            double amount =
                Double.parseDouble(request.getParameter("amount"));
            double interest =
                Double.parseDouble(request.getParameter("interest"));
            int period = Integer.parseInt(request.getParameter("period"));
        %>
        <b>Pincipal using compound interest:</b>
        <%= calculate(amount, interest, period) %>
        <br/><br/>
        <jsp:include page="footer.jsp"/>
    </body>
</html>
```

# Example

## Inventory



# Inventory

## ListServlet

```
package edu.albany.mis.goel.servlets;

import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import javax.sql.RowSet;
import sun.jdbc.rowset.CachedRowSet;
public class ListServlet extends HttpServlet {
    public void init(ServletConfig config) throws
        ServletException {
        super.init(config);
    }
    public void doPost(HttpServletRequest req,
                       HttpServletResponse res)
        throws ServletException {
        doGet(req, res);
    }
}
```

```
public void doGet(HttpServletRequest req, HttpServletResponse
res)
    throws ServletException {
try {
    // Load the driver class
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    // Define the data source for the driver
    String sourceURL = "jdbc:odbc:inventoryDB";
    RowSet rs = new CachedRowSet();
    rs.setUrl(sourceURL);
    rs.setCommand("select * from item");
    rs.execute();
    req.setAttribute("rs", rs);
    getServletContext().getRequestDispatcher("/List.jsp").
        forward(req, res);
} catch(Exception ex) {
    throw new ServletException(ex);
}
}
```

# Inventory

## EditServlet

```
package edu.albany.mis.goel.servlets;

import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.sql.DriverManager;
import javax.sql.DataSource;
import javax.sql.RowSet;
import sun.jdbc.rowset.CachedRowSet;

public class EditServlet extends HttpServlet {
    public void init(ServletConfig config) throws
        ServletException {
        super.init(config);
    }
    public void doPost(HttpServletRequest req,
        HttpServletResponse res)
        throws ServletException {
        doGet(req, res);
    }
}
```

```
public void doGet(HttpServletRequest req, HttpServletResponse
    res)
        throws ServletException {
    try {
        // Load the driver class
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        // Define the data source for the driver
        String sourceURL = "jdbc:odbc:inventoryDB";
        RowSet rs = new CachedRowSet();
        rs.setUrl(sourceURL);
        rs.setCommand("select * from item where id = ?");
        rs.setInt(1, Integer.parseInt(req.getParameter("id")));
        rs.execute();
        req.setAttribute("rs", rs);
        getServletContext().getRequestDispatcher("/Edit.jsp").for-
        ward(req, res);
    } catch(Exception ex) {
        throw new ServletException(ex);
    }
}
```

# Inventory

## UpdateServlet

```
package edu.albany.mis.goel.servlets;

import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import javax.naming.InitialContext;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class UpdateServlet extends HttpServlet {
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }
    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException {
        doGet(req, res);
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException {
        Connection con = null;
        try {
            // Load the driver class
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            // Define the data source for the driver
            String sourceURL = "jdbc:odbc:inventoryDB";
            // Create a connection through the DriverManager class
            con = DriverManager.getConnection(sourceURL);
            System.out.println("Connected Connection");
            PreparedStatement stmt= con.prepareStatement
                ("update item " + "set name = ?, " + "description = ?, " + "price = ?, "
                + "stock = ? " + "where id = ?");
            stmt.setString(1, req.getParameter("name"));
            stmt.setString(2, req.getParameter("description"));
            stmt.setDouble(3, Double.parseDouble(req.getParameter("price")));
            stmt.setInt(4, Integer.parseInt(req.getParameter("stock")));
            stmt.setInt(5, Integer.parseInt(req.getParameter("id")));
            stmt.executeUpdate();
            stmt.close();
            getServletContext().getRequestDispatcher("/List").
            forward(req, res);
        } catch(Exception ex) {
            throw new ServletException(ex);
        } finally {
            try {
                if(con != null) {
                    con.close();
                }
            } catch(Exception ex) {
                throw new ServletException(ex);
            }
        }
    }
}
```

# Inventory

## DeleteServlet

```
package edu.albany.mis.goel.servlets;

import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import javax.naming.InitialContext;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class DeleteServlet extends HttpServlet {
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }
    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException {
        doGet(req, res);
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException {
        Connection con = null;
```

```
try {
    // Load the driver class
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    // Define the data source for the driver
    String sourceURL = "jdbc:odbc:inventoryDB";
    // Create a connection through the DriverManager class
    con = DriverManager.getConnection(sourceURL);
    System.out.println("Connected Connection");
    // Create Statement
    PreparedStatement stmt =
        con.prepareStatement("delete from item where id = ?");
    stmt.setInt(1, Integer.parseInt(req.getParameter("id")));
    stmt.executeUpdate();
    stmt.close();
    getServletContext().getRequestDispatcher("/List").forward(req, res);
} catch(Exception ex) {
    throw new ServletException(ex);
} finally {
    try {
        if(con != null) con.close();
    } catch(Exception ex) {
        throw new ServletException(ex);
    }
}
```

# Inventory

## CreateServlet

```
package edu.albany.mis.goel.servlets;

import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import javax.naming.InitialContext;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class CreateServlet extends HttpServlet {
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }
    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException {
        doGet(req, res);
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException {
        Connection con = null;
        try { // Load the driver class
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            // Define the data source for the driver
            String sourceURL = "jdbc:odbc:inventoryDB";
            // Create a connection through the DriverManager class
            con = DriverManager.getConnection(sourceURL);
            System.out.println("Connected Connection");
            PreparedStatement stmt = con.prepareStatement
                ("insert into item " + "(name,description,price,stock) " +
                 "values (?, ?, ?, ?)");
            stmt.setString(1, req.getParameter("name"));
            stmt.setString(2, req.getParameter("description"));
            stmt.setDouble(3, Double.parseDouble(req.getParameter("price")));
            stmt.setInt(4, Integer.parseInt(req.getParameter("stock")));
            stmt.executeUpdate();
            stmt.close();
            getServletContext().getRequestDispatcher("/List").forward(req, res);
        } catch(Exception ex) {
            throw new ServletException(ex);
        } finally {
            try {
                if(con != null) con.close();
            } catch(Exception ex) {
                throw new ServletException(ex);
            }
        }
    }
}
```

# Inventory

## Edit.jsp

```
<%@page contentType="text/html"%>
<jsp:useBean id="rs" scope="request" type="javax.sql.RowSet" />
<html>
<head>
<title>Inventory - Edit</title>
</head>
<body style="font-family:verdana;font-size:10pt;">
<%
if(rs.next()) {
%>
<form action="Update">
<input name="id" type="hidden" value="<%= rs.getString(1) %>">
<table cellpadding="5" style="font-family:verdana;font-size:10pt;">
<tr>
<td><b>Name:</b></td>
<td>
<input name="name" type="text" value="<%= rs.getString(2) %>">
</td>
</tr>
<tr>
<td><b>Description:</b></td>
<td>
<input name="description" type="text" value="<%= rs.getString(3) %>">
</td>
</tr>
```

```
<tr>
<td><b>Price:</b></td>
<td>
<input name="price" type="text" value="<%= rs.getString(4) %>">
</td>
</tr>
<tr>
<td><b>Stock:</b></td>
<td>
<input name="stock" type="text" value="<%= rs.getString(5) %>">
</td>
</tr>
<tr>
<td></td>
<td>
<input type="submit" value="Update"/>
</td>
</tr>
</table>
<%
}
```

```
%>
</body>
</html>
```

# Inventory

## Edit.jsp

```
<%@page contentType="text/html"%>
<jsp:useBean id="rs" scope="request" type="javax.sql.RowSet" />

<html>
<head>
<title>Inventory - List</title>
</head>
<body style="font-family:verdana;font-size:10pt;">
<table cellpadding="5" style="font-family:verdana;font-size:10pt;">
<tr>
<th>Name</th>
<th>Description</th>
<th>Price</th>
<th>Stock</th>
<th></th>
<th></th>
</tr>
<%
while(rs.next()) {
%>
<tr>
<td><%= rs.getString(2) %></td>
<td><%= rs.getString(3) %></td>
<td><%= rs.getString(4) %></td>
<td><%= rs.getString(5) %></td>
<td>
<a href="Delete?id=<%= rs.getString(1) %>">
Delete
</a>
</td>
<td>
<a href="Edit?id=<%= rs.getString(1) %>">
Edit
</a>
</td>
</tr>
<%
}
%>
</table>
<a href="New.html">New Item</a>
</body>
</html>
```

# Inventory

## New.html

```
<html>
<head>
  <title>Inventory - Add New Item</title>
</head>
<body style="font-family:verdana;font-size:10pt;">

<form action="Create">
<table cellpadding="5" style="font-family:verdana;font-size:10pt;">
<tr>
  <td><b>Name:</b></td>
  <td><input name="name" type="text"/></td>
</tr>
<tr>
  <td><b>Description:</b></td>
  <td><input name="description" type="text"/></td>
</tr>
<tr>
  <td><b>Price:</b></td>
  <td><input name="price" type="text"/></td>
</tr>
<tr>
  <td><b>Stock:</b></td>
  <td><input name="stock" type="text"/></td>
</tr>
```

```
<tr>
  <td></td>
  <td><input type="submit" value="Create"/></td>
</tr>
</table>
</body>
</html>
```