**Java:** Java is a high-level, compiled, strongly typed object-oriented programming (OOP) language. Java

is write once run anywhere programming language (**WORA**). Java is **safe and easier** to use.

Java is **open source** and **free**. Java has many **library and frameworks**. **Automatic memory**

**management**

**WORA**: Platform independent, not constraint to single operating system

**JVM**: Java Virtual Machine reads the compiled java bytecode and converts it into machine code. It

executes code line by line

**JRE**: JRE provides the class, libraries & other resources to run the Java program. The JRE contains JVM

in it. So, to run Java we need JRE

**JDK**: JDK develops and executes Java programs. The JDK also has a JRE inside of it, so if you install a JDK

you can compile your Java code as well as execute it.

**Stack Memory**: used to store items with short life. Temporary memory. Ex: variables & reference of

objects. Stack memory contains reference to the object.

**Heap Memory**: used to store global variable and objects. Heap is the run time memory and store global variables

**Variable:** In java variable are used to store data. Variables are stored in memory. Two Types: **Primitive**

and **Reference**

**Primitive**: byte-8, short-16, char-32, int, long, float, double, Boolean

**Reference**: Object, String, Wrapper classes, Arrays

**Class**: blueprint or templates used to instantiate the object. Contains variables and methods

**Object**: entity with state and behavior. Create by instantiation of class. Used new keyword to create

**Method**: reusable block of code that can take in arguments and can return something (datatype or

object)

**Ex (Signature): accessModifier returnType Name (Parameter) throws … {}**

**Constructor**: type of method as a class name that has no return type and used to create an instance of

a class (object). When the new object is created, the constructor is run. Constructor is used to

initialize instance variables.

**Types**: default, no args, parameterized(overloaded)

**Constructor Changing:** The process of calling one constructor from another constructor is called

constructor chaining. Or the way of calling the constructor of parent class from the constructor

if child class is called constructor chaining. Default no args constructor call super class

constructor.

**OOP**: OOP is the concept where java code can be break down into smaller pieces called classes and

Object where possible. There are four pillars of OOP:

**Inheritance**: The ability for entity to adopt the variable (fields) and methods (behavior) of super class.

Can be done by extending classes or implementing interface. Help to code reuse, reduce

duplication Ex:

**Polymorphisms**: Ability for code structure to be treated as different form during the compilation or the

run time. Can be done by method overloading or method overriding

Ex: Mobile Phone: (phone, calculator, play game, watch video)

**Method Overloading**: creating multiple methods in the same class with the same method name

and different parameters. Example of compile time Polymorphisms. Can not be perform by

changing return type

**Method Overriding**: method in sub class with the same method signature as the parent class but the different implementation. Example of run time polymorphism.

**Encapsulations**: It is the process of restricting the access to the abstracted code by wrapping the code

and data together. It can be done by providing access modifiers. Ex:

**Public**: least restrictive, all the class within the package have access

**Protected**: with in the class in the same package and sub classes have access

**Default**: All the class within the same package have access

**Private**: most restrictive, only the class have access. Can access from other class using getter

and setters

**Abstraction**: Process of hiding the implementation details and showing the function only. It is the way

of making the simple thing into complex. Can be achieve by abstract keyword or interface.

Abstract class cannot be instantiated. May have abstract and non-abstract method

**Interface**: Interface are the special type of class with the abstract methods. It declares the behavior for

classes to implement. All the variables are public static final. Can not be instantiated. Can

implement as many interfaces we want

**Marker Interface**: interface with no declare method and used to give information to the complier

**Functional Interface**: Interface with only one abstract method declared

**Arrays**: An array is a special type of object that act as the collection of same type of data. Array use

special memory in the heap, and it has fixed size.

String [] name = {abc, xyz}; or String [] name = new String [2]

**Varargs**: Variable argument list is a feature that allows you to pass an arbitrary number of values

and treat the data as single array. If used, varargs must be the last parameter and only one

varargs parameter can be used.

**Public void methodName (String name, varargs … name2){}**

**Scope of variables:**

**Instance:** Class level variables (lifetime), **Method:** Variables in method **Block level:**

**Static Keyword**: Static means the methods, or the variable belongs to the class instead of object. It can

invoke directly form class. Static method can be call from other static method only.

**Final Keyword**: Declares a resource (class, method, or field) as the last implementation, which cannot

be extended, overridden, or changed.

**Abstract Keyword**: Allows for the declaration of a class or method without implementation.

**Synchronized:** Limits the number of threads that can access a resource at a given time to 1.

**Wrapper Class**: Wrapper class allow us to treat primitive like objects. Ex: Integer, Float etc

Autoboxing:

Unboxing:

**Upcasting**: Instantiating by declaring parent class reference type

Animal mammal = new Dog ()

**Down casting**: Instantiating by declaring child class reference type

Dog dog = (Dog) (new Animal ())

**String**: array of characters in java. Strings are the immutable class in java. Strings are usually stored in

the string pool(heap) in the memory.

**String Manipulation**: String manipulation is the process of handling and analyzing the string. It is the

way of modifying and changing string data. Ex: compare (), concat (), equals (), Length (), etc.

**StringBuilder & String Buffer**: mutable alternatives to strings with methods that allow for manipulation (append, replace etc.)

StringBuilder is not synchronized (thread safe) -faster

String Buffer is synchronized (thread safe) – slower

**Generic in Java**: Generics are the methods, set of similar types. Generics allow Types, Integer, String or

user defined types to passed as parameter.

**Data Structure**: solution with interact or work with data

**Collection**: Implements Iterable. Collection is the interface that can be considered the “super” interface

from which list, set, queue come from. Work only with wrapper classes(objects).

**List**: Interface that implements collection and can hold duplicate values. List is an ordered list. We can add, change, and remove element anywhere in the list.

**Array List**: Dynamically sized array where element can be added or removed. Elements

can be accessed by index.

**Linked List**: Consists of a series of nodes which store data and a reference to the next

node

**Set**: Interface that extends collection and do not hold duplicate values. Elements are not in

ordered and cannot be find by index. Can add and remove elements.

**Queue**: Interface that extends collection and stored element at the end. Elements are ordered.

All the elements are inserted at the end and remove from beginning. Cannot find out by

the index.

**Map**: Interface that does not implements collection, leverage a key value pair system. They do not implement iterable interface, can not use for each loop directly on map

**Comparable & Comparator**: both are interface and can be used to sort the collection of elements

**Comparable**: Single sorting sequence (sort the collection based on Id, name etc.) This affects the

original class while sorting. CompareTo () method is used. Ex: collection.sort(List).

**Comparator**: Multiple sorting sequence (sort the collection based on multiple elements). Does

not affect the original class while sorting. Compare() method is used. Ex: collection.sort(List,

comparator)

**Functional Interfaces**: Functional interfaces are the interfaces that have only one abstract method. The

purpose of functional interface is to provide a specific function that can be applied easily to any

class.

Ex: **Runnable, Comparable, Comparator** etc.

**Lambdas**: The Lambda expression is one of the biggest new features of Java 8 and introduces aspects

important to the use of functional programming in java. Lambdas allow for the creation and

execution of a function or method without needing to create a dedicated space in memory

**Garbage Collection**: Garbage collection is the process of removing objects from the heap which have no

references to them. There is no way to explicitly force garbage collection in Java; however,

garbage collection can be requested using one of the following:

system.gc(), .runfinalize(), System.runFinalize()

**Finalize**: Finalize is used to perform clean up processing just before object is garbage collected. Finalize is the method, finally is the block and final is the keyword

**Reflections API:** The Reflections API enables Java code to discover information about the fields, methods

and constructors of classes which have been loaded into memory. The API will ‘reflect’ meta data about the fields, methods, and constructors to unveil their underlying information within security restrictions.

**Reflection API**: includes class, constructor, method, field, modifier, parameter

**Class Feature**: getMethod(), getConstructor(), getParameter(), getDeclaredFields() etc.

**Constructor Feature**: getName(), getModifiers(), getParameterName(), getParameterType() etc.

**Method Feature**: getName(), getParameterType(), getParameterName(), getReturnType() etc.

**Field feature:** getModifiers(), getType(), getByte(), getDouble() etc.

**Threads:** A thread is a subset of a process that is also an independent sequence of execution.

**Synchronization**: synchronized keyword which prevents more than one thread at a time to access a

resource

**Multi-Threading**: Multithreading extends the multitasking of applications by subdividing operations in a

single application into individual, parallel threads

**Creating Thread**:

Create a class the implements the runnable functional interface

Create a class that extends Thread

Create a Lambda\*

Override the run method

**States of Thread**:

**New**: Created a thread object but run has not been called

**Runnable**: Run has been called and been executing

**Waiting**: stops the execution of a thread until another thread is done executing

**Blocked**: waiting for another thread to be done accessing a resource on which a lock has

been placed on

**Terminated**: Everything has been executed

**POJO:** Plain Old Java Object. Any java object not bound to any specific restriction Used to define an entity

**Java Bean**: Special type of POJO. All java beans are POJO but all POJO are not Java beans. Should have

no-args constructor, all fields should be private, getters and setters, override toString, equals and

hascode method.

**Exception**: Exception are the problems that occurs during the execution of the java code. Or exception

is a condition that prevents the method for completing successfully. Exceptions **inherits from**

**Throwable** class.

**Types of Exception**:

**Checked Exception**: checked exceptions are checked by compiler. Can not compile code until

exception is handled

Ex: FileNotFoundException, ClassNotFoundException etc.

**Unchecked Exception**: Subclasses of Runtime Exception. Arises from coding error. Occurs in

Runtime

ArthmeticException, ArrayOutOfBoundException etc.

**Exception Handling**:

**Try Block**: used to execute the code or statement that may throw exception

**Catch Block**: Exception handler. Used to catch the exception

**Finally Block**: Always execute regardless of outcomes. Used to close resources of try block

**Try-With-Resources**: allows resources to close without needing finally block

**Throws**: used in method signature and used to throw exception. The exception will

handled in method where it is called

**Throwing Exception**: throw new Exception(“messages”)

**Custom Exception**: can create own exception by extending Exception or its sub class

**Error**: Errors are irrecoverable problems in java occur at compile time. Ex: out of memory

**Maven**: Maven is the project management and build automation tool which create artifact to be

deployed and manages the lifecycle of an application. Maven is used for properties and

dependency management also.

**lifecycle**: maven clean, maven compile, maven deploy

**dependency**: Frameworks or libraries written by other developers to be used in our project.

**Testing**: Testing is the process of making sure that the application is working as expected

**Positive Testing**: Making sure application handles correct inputs

**Negative Testing**: Making sure application handles incorrect inputs

**Regression Testing**: Making sure new features added to the application do not break existing

Features

**Testing Types (Pyramid)**: Unit Testing, Integration Testing, System Testing, User Acceptance Testing

**Unit Testing**: Unit testing is the testing of the individual software components in isolation from the rest

of the system. In simple word unit testing is the testing of the method of the code

**Junit**: Junit is an open-source java framework designed for the purpose of writing and running the unit

Testing. Junit allows for the creation of application code and run during testing.

**Junit Annotations**:

@BeforeClass: declare set up method that run once

@Before: run before every test are run

@Test: declare method as a test method

@After: run after every test are run

@AfterClass: declare tear-down method, run at the end of the test

**Junit Assertions**:

AssertEquals(), AssertNotEquals(), AssertTrue(), AssertFalse(), AssertNull(), AssertNotNull() etc.

**Test Driven Development**: The TDD process consists of writing unit tests first before the application code

has been written. Then, code can be written to make the test pass, and the process can be

completed for each piece of functionality required

**TDD Process**:

Write a unit test

Run the test: test will fail

Fixed the text by writing application code

Rerun the test

**Logging**: Logging refers to the act of recording granular events within an application, such as transactions

with a database, exceptions or debugging events

**Log4j2**: Log4j is a reliable, fast, and flexible logging framework for Java supported by Apache. It is

commonly used to record application events and write them to a file

**Main Components**:

**Logger**: logs the messages

**Appender**: publishes logs to destination(s)

**Layout**: formats logging information

**Configuration**: stores settings

**Filter**: used to filter logs that do not meet some threshold or are not required

**Logging Levels**:

**ALL:** all levels

**TRACE:** finer-grained informational events than DEBUG

**DEBUG:** designates informational events that are most useful to debug an application

**INFO:** informational messages that highlight the progress of the application at the coarse-

grained level

**WARN:** designates potentially harmful situations

**ERROR:** designates error events that might still allow the application to continue running

**FATAL:** severe error events that presumably lead the application to abort

**OFF:** highest possible level, intended to turn off logging

**HTML & CSS**

**HTML:** stands for Hypertext Markup Language - it is a markup language for creating web pages and applications.HTML contains a particular syntax - namely **elements** and **attributes** - that web browsers parse to render the content of the webpage.

**HTML5:** HTML5 introduced a new **DOCTYPE** declaration <!DOCTYPE html> and the character encoding (charset) declaration <meta charset="UTF-8">. The <DOCTYPE> declaration is used to inform the browser about the version of HTML used in the document. It is known as the Document Type Declaration (DTD). It just instructs the browser about the document type. A **character encoding** is an approach of converting bytes into characters. For validating the HTML document, a program must choose a character encoding.

HTML5 also introduced features to allow us to embed audio and video files on the web page and provides the support to run JavaScript in the background.

**HTML5 Semantic Elements**

* The HTML elements like <div> and <span> are not releated to the content on the web page. This kind of elements are called as non-semantic elements.
* The HTML elements like <form>, <table>, and <article> are used to define the content and on the webpage. This kind of elements are called as semantic elements.

<!DOCTYPE html>

<html>

<head>

<title>Hello World!</title>

<meta charset="utf-8">

</head>

<body>

<div>

<!-- THIS IS A COMMENT! -->

<p>This is my first paragraph written in HTML</p>

</div>

</body>

</html>

## **Elements**

HTML is composed of many different **elements** - these provide the structure of the document. Elements are defined within HTML files using **tags** - for example, one very common tag is the <div> tag.

## **Attributes**

HTML elements can also have **attributes** defined within the tag - these are key/value pairs that give metadata about the tag that are important for the browser to know. Ex: <img src = “URL”>

### **Global Attributes:** class, id, name, style, title, hidden, required

## **Inline and Block Elements**

Before listing all the HTML elements available to use, it's important to know the difference between inline and block-level elements.

* Block-level elements are those that will render on new lines in blocks by default, instead of rendering within the line itself like inline elements do. One example of a block element is <div>
* An inline element does not start on a new line. <span>

## **Common HTML Tags**

There are a vast number of HTML tags you could use on your webpage, but below are listed the most common:

* <div> - defines a "division of the page"
* <p> - defines a paragraph
* <span> - an inline tag for grouping text or elements
* <b> - bold text
* <i> - italicized text
* <h1>, <h2>, ... <h6> - these are headings, h1 is largest and h6 is smallest
* <br> - line break
* <table> - defines a table
* <img src="URL">
* <ol> - an ordered list
* <ul> - an unordered list
* <a href=""> - makes a hyperlink

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

</tr>

</thead>

<tbody>

<tr>

<td>1</td>

<td>Alice</td>

</tbody>

</table>

<nav>

<a href="/html/">HTML</a> |

<a href="/css/">CSS</a> |

<a href="/js/">JavaScript</a> |

<a href="/jquery/">jQuery</a>

</nav>

# HTML Forms

An **HTML form** is a section of a document that contains controls such as text fields, password fields, checkboxes, radio buttons, submit button, menus, etc. Using these elements, the page can collect information from a user which is typically submitted to a web server. To create a form, you would use the <form> tag.

**Attributes Used in HTML Forms**

There are several attributes that you can use on the <form> tag and on <input> elements.

* **Action**: Action attribute indicates where the form data will be processed. The value is a URL of a server.
* **Target**: Where to open a browser. New browser or the current browser
* **Name**:
* **Method**: GET or POST
* **Value**:
* **Placeholder**:
* **Required**:

<!DOCTYPE html>

<html>

<body>

<form action="/test.php" target="\_blank" method="GET">

Username:<br />

<input type ="text" name="username" placeholder="Username" required/>

<br />

Password:<br />

<input type ="password" name="password" />

<br /><br />

<input type ="submit" value ="Submit" />

</form>

</body>

</html>

# Input Element in HTML Forms

<input type ="text" name="username" placeholder="Username" required/>

<input type="text" name="email-input" />

<input type="password" name="user-password"/>

<input type="checkbox" name="subject" id="math" />

<input type="radio" name="gender" id="male">

<input type="submit" value="Submit" />

# HTML5 Validation

**Built-in form validation** - It uses HTML5 form validation features. (required, min length, max length, pattern for email, min character, max character etc)

**JavaScript validation** - It is coded using JavaScript. This validation is completely customizable.

# Overview of CSS

CSS stands for **Cascading Style Sheets** - it is a language for styling HTML documents by specifying certain rules for layout and display in key/value pairs. Style Sheets are a simple and powerful method of allowing attachment of rendering information to HTML documents. It used to style the webpages by setting background-color, font color, font size, font family, etc.

A CSS consists of a set of rules that defines the styles for a web page. A CSS style rule composed of **selectors** and **declarations**. The selector is an HTML Element like h3 used in the below example. The declaration is comprised of a property and a value surrounded by curly braces. In the below example font-family, font-style and color were properties of the selector h3. Arial, italic and red were the values assigned, respectively, to the properties.

h3 {

font-family: Arial;

font-style: italic;

color: red

}

# CSS Box Model

Every box has 4 parts - **margin**, **border**, **padding** and **content**. The margin is an outermost box, inside that the border, then padding, then the content is innermost. All box sizes/formatting can be styled with CSS.

**Margin** - It is a space between border and margin. It is useful to separate the element from its neighbors. The dimensions are given by the margin-box width and the margin-box height.

**Border** - It is the area between the box’s padding and margin. Its dimensions are given by the width and height of the border.

**Padding** - It is a space around the content area and within the border-box. Its dimensions are given by the width of the padding-box and the height of the padding-box.

**Content** - It consists of content like text, image, or other media content. It is bounded by the content edge and its dimensions are given by content-box width and height

# Types of CSS

There are three types of CSS which are given below:

* Inline CSS
* Internal or Embedded CSS
* External CSS

### **Inline CSS**

Inline CSS contains the CSS property in the body section attached with element is known as inline CSS. This kind of style is specified within an **HTML tag using style attribute**.

<body>

<p style = "color:#009900;

font-size:50px;

font-style:italic;

text-align:center;">

Hello World

</p>

</body>

### **Internal or Embedded CSS**

This can be used when a single HTML document must be styled **uniquely**. The CSS rule set should be within the HTML file in the head section i.e the CSS is embedded within the HTML file.

### **External CSS**

External CSS contains separate CSS file which contains only style property with the help of tag attributes (For example class, id, heading, … etc). CSS property written in a separate file with .css extension and should be linked to the HTML document using **link** tag. This means that for each element, style can be set only once and that will be applied across web pages.

**Example:** The file given below contains CSS property. This file should be saved with an .css extension. For Ex: **style.css**

body {

background-color:powderblue;

}

.main {

text-align:center;

}

.hi {

color:#009900;

font-size:50px;

font-weight:bold;

}

# CSS Properties

There are a number of CSS properties that you can use to style our webpage. Here we'll discuss some of the CSS properties such as Border, Padding, Margin, display, position, color,and text-align.

### **CSS Border Property**

The CSS border property allows to style the border area of a box. The properites and corresponding vlaues with examples covered under the CSS border are tabulated below:

| **Property** | **values** | **Usage** | **Example** |
| --- | --- | --- | --- |
| border-width | medium, thin, thick, length | Used to define the border area of a box | div { border-width: medium 10px thick 15px; } |
| border-style | none, hidden, dashed, dotted, double, groove, inset, outset, ridge and solid | sets the style of a box's border | p { border-style: dotted; } |
| border-color | hex-value for colors | specify the color of a box's border | p { border-style: solid; border-color: #ff0000; } |

### **CSS Padding Property**

The CSS padding property allow you to set the padding area for an element that separates its border from its content. The padding property can take one, two, three, or four values separated by white spaces as listed in the below table. Depending on the list of property values, the HTML element has the padding area on the top, bottom, right, and left.

| **Examples** | **Explanation** |
| --- | --- |
| p { padding: 70px; } | Sets the padding for an <p> element to 70 pixels for all four sides |
| p { padding: 35px 70px; } | Sets the padding for an <p> element to 35 pixels for top and bottom and 70 pixels for right and left sides. |
| p { padding: 35px 70px 40px; } | Sets the padding for an <p> element to 35 pixels for the top, 70 pixels for the left and right side and 40 pixels for the bottom. |
| p { padding: 35px 70px 40px 80px; } | Sets the padding for an <p> element to 35 pixels for the top, 70 pixels for the right side, 40 pixels for the bottom and 80 pixels for the left side. |

The padding property is a shorthand property for the padding-top, padding-right, padding-bottom, and padding-left properties. The below examples set padding on a specific side for the HTML element.

Examples:

h1 {

padding-bottom: 10px;

}

p {

padding-top: 20px;

padding-left: 50px;

}

### **CSS Margin Property**

The CSS margin property is similar to the CSS border property, but it sets the margins around the sides of an element's box instead of the border. It also takes one, two, three, or four values separated by white spaces. The shorthand properties are margin-top, margin-right, margin-bottom, and margin-left to set a margin on respective sides.

Example:

p {

margin-left: 10px;

margin-right: 30px;

}

h1{

margin: 25px 50px;

}

### **CSS Display Property**

The display property controls the display behaviour of an element. The CSS display property sets whether an element is treated as a block or inline elements and the layout used for its children, such as flow layout, flex or grid.

There are two types of HTML elements: **inline-level elements** and **block-level elements**. The differences between these elements affect how you use the box model. Both Inline and block-level elements appear within the body of an HTML page. But, inline-level elements are used to create a short structure that can have data and other inline elements. Inline level elements include <b>, <big>,< i>, <small>, <tt>, <abbr>, <acronym>, <code>, <strong>, etc.  
Block-level elements used to create larger structures than inline elements also it starts on new lines by default whereas inline-level elements not. Block elements include <p>, <h1>, <h2>, <h3>, <h4>, <h5>,<h6>, <ol>, <ul>, <pre>, <address>, <blockquote>, <dl>, <div>, <fieldset>, <form>, <hr>, <table>, etc .

The Syntax for the display property is selector {display: value}. The property values and description with examples are tabulated below:

| **Property value** | **Description** | **Example** |
| --- | --- | --- |
| block | behaves likes block-level elements | a {display: block;} |
| inline | behaves like inline-level elements | ul li { display: inline; } |
| none | elements doesn't generate boxes | h1 { display: none;} |

### **CSS Position Property**

The position property defines how an element will be arranged on a page. The Syntax for the position property is selector {position: value}. The property values are static, relative, absolute, fixed, or inherit.

static - The element's box is arranged automatically consistent with the normal flow.

relative - The element's box position is relative to its normal flow position. You can adjust the normal flow position by using the top, bottom, left and right properties.

absolute - The element's box arranged to an absolute position with reference to its containing block. Its containing block is that the nearest ancestor element that has its position property set to relative, absolute, or fixed. The top, right, bottom, and left properties are used to set the offset of the element's box with reference to its containing block.

fixed - The element's box position is offset from its browser window by using the top, right, bottom, and left properties. The element's box won't move when the browser window is scrolled.

inherit - The inherit keyword is employed to specify that the value for this property should be taken from the parent element. If inherit is used with the root element, then the initial value for this property is going to be used.

Example:

a {position: static;}

div {position: relative; top: 20px; left: 50px;}

h1 {position: absolute; top: 30px; left: 20px;}

div {position: fixed; top: 325px; left: 60px;}

### **CSS Color property**

The color property is used to specify the foreground color of text.The color properties are set using 5 different color notation types which is listed below:

a {color: red;}

div {color: #3c5;}

h1 {color: #ffa500;}

div {color: rgb(100,20,255);}

#id1 {color: rgb(30%,50%,70%);}

### **CSS text-align property**

The text-align property is used to align the content inside the element. The text inside the element can be aligned in 4 ways - left, right, center and justify.

Example: The text-align properties are set to left, right, justify, and center.

div {text-align:left;}

h1 {text-align: right;}

p {text-align: justify;}

div {text-align: center;}

## **Element Selector**

The element selector selects HTML elements by their name / tag name like a, h1, div, p etc.

Example: Here, we use <p> as an element selector. The text inside the <p> will be center-aligned also blue color.

## **Class Selector**

In the CSS, the class selector is a name preceded by a period (“.”). It uses the class attribute of an HTML element to match the specific HTML element. We can have a Class selector specific to an HTML element like we have *p.class* in the below example.

## **ID Selector**

In the CSS, the ID selector is a name preceded by a hash character (“#”). It uses the id attribute of an HTML element to match the specific HTML element. The **id** of an element should be unique within a page, so the id selector is used to select one unique element.

# Responsive Web design

Responsive Web design is the approach that allows websites and pages to render (or display) on all devices and screen sizes by automatically adapting to the screen, whether it’s a desktop, laptop, tablet, or smartphone. Responsive web design works through CSS, using various settings to serve different style properties depending on the screen size, orientation, resolution, color capability, and other characteristics of the user’s device.

# CSS3 Media Queries

CSS3 supports responsive web design, all kinds of transitions, transformations, and animations and provides box-sizing tools that enable the user to adjust the size of any element without changing the dimensions or padding of the element.

Media queries allow you to customize the presentation of your web pages for a specific range of devices like mobile phones, tablets, desktops, etc. without any change in markups. It composed of a media type and expressions that check for the conditions of particular media features. It is a logical expression that is either true or false.

# Bootstrap

Bootstrap is an open-source framework and mobile-first approach for developing responsive websites. It is a front-end framework programmed to support both HTML5 and CSS3. It comprises the list of components such as Typography, Code, Table, Forms, Button, Images, Icons, etc. A responsive website is a website that automatically adjusts the screen size and looks good on all devices, from smartphones to a desktop. It is easy to use, saves time and customizable. Bootstrap 4 is the newest version of Bootstrap.

### **Client and Server Architecture**

A client-server architecture is a networking model in which the server provides services to clients to perform user-based tasks.

**Server** - A server is software designed to process requests and deliver responses to another computer over the internet.

**Client** - A client is a program that runs on a local machine requesting service from the server.

A Client and a Server establish a connection according to a set of rules called a protocol. There are quite a few protocols for different purposes, but one of the most popular is the **HTTP protocol**. Once the connection is established, the Client sends **HTTP Requests** to the server in the form of XML or JSON, which both entities (Client and Server) understand. After parsing the request, the Server responds with appropriate data by sending back an **HTTP Response**.

### **Types of Client-Server Architecture**

**2 tier architecture** - The user interface stored at the client machine and the database stored on the server. If Business Logic & Data Logic collected at a client-side, then it is known as a fat client thin server architecture. If Business Logic & Data Logic handled on the server, then it is known as a thin client fat server architecture. 2 tier architecture has some limitations in performance, security, and portability.

**3 tier architecture** - Three-tier architecture has a middleware between the user interface and database. The 3 tiers are named the presentation tier, application tier, and data tier. The presentation tier is the front end layer and consists of the user interface. The application tier contains the functional business logic which drives an application’s core capabilities. The data tier consists of a database system and the data access layer.

**n-tier architecture** - In n-tier architecture, there are multiple Business Logic & Data Logic layers. It increases the flexibility and reusability of applications but can be difficult to implement.

## **Servlets**

A website can consist of both static and dynamic webpages. A static webpage is a pre-built HTML page with the content explicitly written into the code, and stored in the webserver. Static web pages display the same content each time we visit. A dynamic webpage loads dynamic content such as stock prices, weather information, news, and sports updates at different points of time. In Java, there exists a way to generate static webpages with dynamic data, and that's with **Java Servlets**.

A **Servlet** is a Java class that takes incoming requests, processes them, and generates a response to send back to the user. For example, an HttpServlet takes an HTTP request, processes its headers and content, and uses that information to write HTML, CSS, and JavaScript code into an HTTP response that can be sent back to the user's browser. The **Servlet container** is the component of an **application server** that interacts with Java servlets and is responsible for managing the execution of servlets and JSP pages for Java applications.

### **How do servlets work?**

When a client sends a request to the application server, the application server receives and passes the request to the appropriate servlet. The servlet processes the request, generates the response, and sends the response back to the application server. The application server sends the response back to the client. Most servlets are HTTPServlets, which receive HTTP requests and generate HTTP Responses out of HTML, CSS, and JavaScript code.

### **Life Cycle of a Servlet**

A **servlet container** manages the life cycle of a servlet. [Servlet](https://docs.oracle.com/javaee/1.4/api/javax/servlet/Servlet.html) is an interface defined in **javax.servlet** package. A servlet container uses the Servlet interface to understand a specific Servlet object and manage it.

There are three life cycle methods of a Servlet :

* init()
* service()
* destroy()

The steps involved in the servlet life cycle are listed below:

**Step-1 : Loading of Servlet**

When the application server (e.g. Apache Tomcat) starts up, the servlet container deploys and loads all the servlet classes.

**Step-2 : Creating an instance of Servlet**

Once all the Servlet classes are loaded, the servlet container creates only one instance for each servlet class. All requests to the servlet are executed on that same servlet instance. Some application servers can create multiple instances of a servlet to handle a high volume of incoming requests, but that is not the default behavior.

**Step-3 : Invoke init() method once**

Once all the servlet classes are instantiated, the init() method is invoked for each instantiated servlet. The init() method is used to initialize the servlet. The init() method is called only once.

**The init() method signature:**

public void init() throws ServletException {

}

**Step-4 : Invoke service() method repeatedly for each client request**

The servlet container calls the service method each time a request for the servlet is received. The service() method determines the type of Http request (GET, POST, PUT, DELETE, etc.) also calls doGet(), doPost(), doPut(), doDelete(), etc. methods as appropriate.

**The service() method signature:**

public void service(ServletRequest req, ServletResponse resp) throws ServletException, IOException {

}

**Step-5 : Invoke destroy() method once**

The destroy() method is called only once at the end of the a servlet's life. The servlet container calls this method before removing the servlet instance from the service.

**The destroy() method signature:**

public void destroy() {

}

Servlet Declarations and Mapping

<servlet>

<servlet-name>servlet1</servlet-name>

<servlet-class>com.revature.MyFirstServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>servlet1</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

## **Deployment descriptor**

Java web applications use a **deployment descriptor file** to define the URLs that map to servlets, and to determine which URLs require authentication and additional information. A deployment descriptor file specifies the classes, resources, and configuration of the application and how the web server uses them to serve HTTP requests.The deployment descriptor is a file named **web.xml**. It resides within the app's WAR beneath the WEB-INF/ directory.The root element of the web.xml file is <web-app>.

The **web.xml** file defines mappings between URL paths and the servlets that will handle requests with those paths.

.

## **Writing plain text to the response Object**

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

{

// set the content type as plain text

response.setContentType("text/plain");

PrintWriter out = response.getWriter();

out.println("Hello World"); // writes a plain text to the response object

}

## **SendRedirect in servlets**

**sendRedirect(String URL)** - This method defined in **HttpServletResponse** interface and used to redirect a response to another resource. It uses the URL to make another request. Therefore, it works at the client-side also can work inside and outside the server.

The RequestDispatcher.forward() method is used to pass the same request to a new destination resource, but the Response.sendRedirect() method is used to send an entirely new request for the destination resource. Any request attributes or parameters from the original request are lost.

**Example for sendRedirect():**

//"response" is a HttpServletRequest Object redirected to the google server.

response.sendRedirect("http://www.google.com");

## **Working with Jackson API**

The [Jackson API](https://en.wikipedia.org/wiki/Jackson_(API)) is used to convert Java objects into JSON format to send in an HTML response. The JSON format is commonly used for transmitting data between a server and a client in a web application.

### **Jackson ObjectMapper**

The ObjectMapper API is used for data-binding. It uses reader/writer methods to perform the conversion between Java objects and JSON.

ObjectMapper mapper = new ObjectMapper();

User user = new User();

//Object to JSON in String

String jsonInString = mapper.writeValueAsString(user);

## **Handling form data from the request**

When a user fills in the fields of a form and submits it, the servlet processes the request based on the submitted data and sends a response back to the client.

* **method="post" or method="get"** : to send the form data as an HTTP POST or HTTP GET request to the server.
* **action= "URL of the servlet"**: specifies URL of the servlet which is responsible for handling form data.

## **Retrieving request parameters from the request**

Servlets use the following methods for retrieving request/form parameters from the HTTP request:

* getParameter() method - used to get the value of a specified parameter.
* getParameterValues() method - used to get the multiple values of a specified parameter.(for example checkbox)
* getParameterNames() method - used to get complete list of all parameters.

## **Request Dispatcher**

The Request Dispatcher interface defines an object that passes along the client's request to any other resources (servlet, JSP file, or HTML file) on the server.

### **1. forward(ServletRequest request, ServletResponse response)**

forward() passes a request from one servlet to another resource on the server. The contents of the request and response are preserved and forwarded to the next resource which will process the data and return the response to the client.

### **include(ServletRequest request, ServletResponse response)**

include() does not entirely transfer control over the request and reponse object to the next resource. Instead, this method includes the content of the original resource in the reponse returned to the client. If you include() a servlet or JSP document, the included resource may not change the response code or HTTP headers.

# REST

Stands for **Represenational State Transfer**.

Defined originally by Roy Fielding in his dissertation in 2000, REST is an architectural style that outlines communication between a client and server over the web. A RESTful server should not retain information about the state of the client.

Clients communicate with the server through an interface that is standard in that it too follows another set of constraints: "defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state." - Roy Fielding

## **API - Application Programming Interface**

An API is a software intermediary that allows two applications to talk to each other.

For example, all airline ticket booking applications will use an API exposed by the airline company. Anytime a customer uses any such application and books a flight ticket, the application passes the passenger and flight booking information to the API. The booking API will process the data and book the ticket for the customer and the application will get a success response with booking details in return. The booking applications do not know and need not to know how the API works internally. All they are required to do is pass the booking information in a well-defined format to the API and wait for the response. Similarly, any application/mobile app can use an API or expose an API to other software.

## **Exception and Error Handling**

The Servlet API allows us to handle exceptions and errors caused during the execution of a servlet and still send a useful response to the user. The **deployment descriptor** file just needs to be configured to handle the exceptions/errors thrown by a servlet.

* <error-code> - used to specify a valid HTTP error code. For example, 404, 403, 500, etc.
* <exception-type> - used to specify a fully-qualified class name of a Java exception type. For example, *javax.servlet.ServletException,java.io.IOException*, *java.lang.RuntimeException*, etc.
* <location> - used to specify the location of the resource which is displayed to the user in case of an error. This might be a servlet, an HTML page, a JSP page, or something else.

**Request Attributes for Errors/Exceptions:**

Before the servlet container invokes the servlet to handle the exception, it sets some attributes in the request to get useful information about the exception. Some of these are:

* javax.servlet.error.status\_code
* javax.servlet.error.servlet\_name
* javax.servlet.error.exception
* javax.servlet.error.request\_uri
* javax.servlet.error.exception\_type
* javax.servlet.error.message

<error-page>

<exception-type>java.lang.ArithmeticException</exception-type>

<location>/errorHandler</location>

</error-page>

## **HttpSession API**

The Servlet API provides **HttpSession Interface**, which provides a way to identify a user and to store information about that user. For the client's first request, the Servlet Container generates a **unique session ID** and gives it back to the client with a response. Thereafter, the client sends the session ID with each request to the server.

The **getSession()** method of the HttpServletRequest object returns a user's session. Any servlet can access the HttpSession object using getSession() method.

Example for creating the HttpSession object:

protected void doPost(HttpServletRequest request,HttpServletResponse response)throws ServletException, IOException {

HttpSession session = request.getSession();

}

The commonly used HttpSession Interface methods are listed below:

* **setAttribute(key,object)** - used to bind an object to the session, using the key specified.
* **getAttribute(String)** - used to retrieve a specific saved object from the session object, using its key.
* **removeAttribute(key)** - used to remove the object bound with the specified key from the session.
* **invalidate()** - destorys the session.
* **getId()** - returns the unique ID assigned to the session.
* **getCreationTime()**- returns the time when the session was created
* **getLastAccessedTime()** - returns the last time the client sent a request associated with the session
* **getMaxInactiveInterval()** - returns the maximum time interval, in seconds.
* **setMaxInactiveInterval(int interval)** - Specifies the time, in seconds,after servlet container will invalidate the session.

## **Session Management in servlets**

The HTTP protocol is a **stateless protocol**, which means no client information stored in the server. The server considers every request form the same client as a new, independent request. However, this means that a server cannot keep a user "logged in" naturally.

A session stores the **unique identification information about the client** that we can get for all requests that client makes. There are four different techniques used by the Servlet application for session management.

* Cookies
* Hidden form fields
* URL Rewriting
* The HttpSession API

## **URL Rewriting**

URL Rewriting is a process by which a **unique session ID gets appended to each request URL**, so the server can identify the user session. URL Rewriting maintains the session and works even the user's browser does not support cookies. This makes it one of the ways in which we can provide a unique id in request and response.

## **Cookies**

A cookie is a key-value pair of information sent by the server to the client, which the client will store. The client (usually a web browser) can send this cookie in the HTTP request header for all subsequent requests until the cookie becomes invalid.

The Servlet container checks the request header for cookies, most commonly to get the session information from the cookie, which it uses it to retrieve the associated session data stored in the server.

* **Non-persistent cookie** - Cookie becomes expired when the user closes the browser.
* **Persistent Cookies** - Cookie expires only if the user logs out of the website. The cookie is stored on the browser even the user closes the browser each time.

### **Creating Cookies with Servlets**

To send cookies to the client, we need to create a Cookie object, set the maximum age for the cookie, and place the cookie in the HTTP response header. The Cookie(String name, String value) constructor defined in the **javax.servlet.http.Cookie** class can be used to create a cookie with a specified name and value. We can use the setMaxAge() method to set the maximum age for the particular cookie in seconds. We can use the response.addCookie() method to place the cookie in the HTTP response header.

### **Reading Cookies with Servlets**

To read cookies, We need to create an array of javax.servlet.http.Cookie objects by calling the getCookies() method of HttpServletRequest. Then the getName() and getValue() methods used to access each cookie and associated value.

## **Hidden Form Fields**

Hidden fields can be inserted into webpages by the server for session tracking.These fields are not visible directly to the user but can still be viewed using the view source option from the browser.

Hidden fields may be used to send information that is only pertinent to the server, and not the client.

<input type = "hidden" name = "session\_id" value = "65349">.

The server retrieves this hidden form field value using the request.getParameter("session\_id") method in a servlet.

## **Front Controller Design Pattern**

The front controller design pattern provides a **single handler** for all the incoming requests for a resource in an application, and then **dispatches** the requests to the appropriate secondary handler for that type of request. The front controller may use other **helper** APIs to achieve the dispatching mechanism.

**Front Controller** - The Front controller is a single-entry point for all requests, and routes incoming user requests. It delegates to a dispatcher to perform action and view management.

**Dispatcher** - A dispatcher is responsible for the action and view management, including locating and routing to the specific actions that will service a request, and finding the appropriate view.

**Helper** - We use Helper classes to break out specific features and make the application easier to build and maintain. They can be used for the retrieval of content, validation of user-entered information, processing of business logic, and data processing.

**View** - A view represents and displays information to the client - think an HTML/CSS/JS page. The view retrieves information from model objects. These model objects can be passed to the view from a Front Controller servlet through a request attribute, or by placement in the web application's session data.

The **benefits** of using the front controller design pattern:

* It provides centralized control for all requests, which helps for user tracking and security.
* It improves manageability, reusability, and role separation.

**ServletConfig and ServletContext parameters**

### **ServletConfig**

ServletConfig is an object created by the Servlet Container, used to pass initial parameters or configuration information to a particular servlet during initialization. The <servlet> XML element in the deployment descriptor (web.xml) has a subelement called <init-param> used to pass parameters to a servlet from the web.xml file.

### **ServletContext**

ServletContext is the object created by the Servlet Container to share initial parameters or configuration information to all servlets and other components. The <context-param> element used to declare the parameters of ServletContext. It present outside the <servlet> element and inside the <web-app> element. This object is returned by the getServletContext() method of HttpServlet.

The <param-name> and <param-value> used to declare the parameter name and its value.

## **MVC Design pattern**

The **M**odel **V**iew **C**ontroller Design pattern is an architectural pattern, used to design and create user interfaces and the structure of an application.This pattern divides the application into three parts that are dependent and connected.

**Model** - Model used to represent the business layer of the application. It's not involved in the UI or presentation of the application. The model is the data layer which defines the business logic of the system and also represents the state of the application. The model objects retrieve and store the state of the model in a database.

**View** - The view presents the model’s data to the user. The view is a presentation layer that represents the user interface. It displays the data fetched from the model layer by the controller and presents the data to the user whenever requests.

**Controller** - A controller is an intermediary between the model & a view. It receives the user requests from the view layer and processes them. The requests are then sent to model for data processing. Once they are processed, the data is again sent back to the controller and then displayed on the view.

**Advantages** - Multiple developers can simultaneously work on the model, controller and views. The MVC pattern enables a logical grouping of related actions on a controller together. The views for a specific model are also grouped. Models can have multiple views.