**Internet Technology**

**Department of Computer Science**

**Milan Kumar Nayak**

**Cont.6370158954**

**Basudev Godabari Degree College**

At /- Kesaibahal , Dist/- Sambalpur , Pin – 768228

[bgdegreecollege@gmail.com](mailto:bgdegreecollege@gmail.com) **,Mob – 9438000807**

**UNIT - JAVA**

The Internet is a worldwide system of interconnected computer networks. The computers and computer networks exchange information using TCP/IP (Transmission Control Protocol/Internet Protocol). The computers are connected via the telecommunications networks, and the Internet can be used for e-mailing, transferring files and accessing information on the World Wide Web.

**Introduction to Object-oriented Programming**

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts −

* Polymorphism
* Inheritance
* Encapsulation
* Abstraction
* Classes
* Objects
* Instance
* Method
* Message Passing

In this chapter, we will look into the concepts - Classes and Objects.

* **Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
* **Class** − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

## Classes in Java

A class is a blueprint from which individual objects are created.

Following is a sample of a class.

### Example

public class Dog {

String breed;

int age;

String color;

void barking() {

}

void hungry() {

}

void sleeping() {

}

}

A class can contain any of the following variable types.

* **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
* **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
* **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Following are some of the important topics that need to be discussed when looking into classes of the Java Language.

## Constructors

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor −

### Example

public class Puppy {

public Puppy() {

}

public Puppy(String name) {

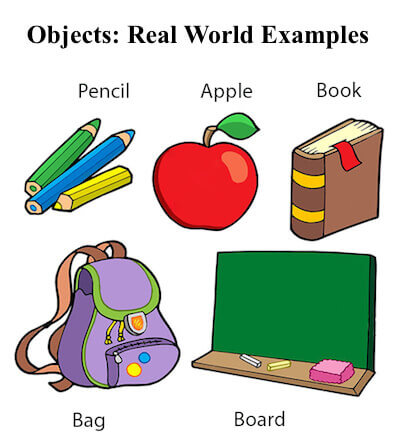
// This constructor has one parameter, *name*.

}

}

Java also supports [Singleton Classes](https://www.tutorialspoint.com/java/java_using_singleton.htm) where you would be able to create only one instance of a class.

**Objects in Java**

In object-oriented programming technique, we design a program using objects and classes.An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

### What is an object in Java

An entity that has state and behavior is known as

an object e.g., chair, bike, marker, pen, table,

car, etc. It can be physical or logical

(tangible and intangible). The example of an

intangible object is the banking system.

An object has three characteristics:

* **State:** represents the data (value) of an object.
* **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
* **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

**Object Definitions:**

* An object is *a real-world entity*.
* An object is *a runtime entity*.
* The object is *an entity which has state and behavior*.
* The object is *an instance of a class*.

**There are many ways to create an object in java. They are:**

* By new keyword
* By newInstance() method
* By clone() method
* By deserialization
* By factory method etc.

## Creating an Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class −

* **Declaration** − A variable declaration with a variable name with an object type.
* **Instantiation** − The 'new' keyword is used to create the object.
* **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object −

### Example

public class Puppy {

public Puppy(String name) {

// This constructor has one parameter, *name*.

System.out.println("Passed Name is :" + name );

}

public static void main(String []args) {

// Following statement would create an object myPuppy

Puppy myPuppy = new Puppy( "tommy" );

}

}

If we compile and run the above program, then it will produce the following result −

### Output

Passed Name is :tommy

## Example: Java Class and Objects

class Lamp {

boolean isOn;

void turnOn() {

// initialize variable with value true

isOn = true;

System.out.println("Light on? " + isOn);

}

void turnOff() {

// initialize variable with value false

isOn = false;

System.out.println("Light on? " + isOn);

}

}

class Main {

public static void main(String[] args) {

// create objects l1 and l2

Lamp l1 = new Lamp();

Lamp l2 = new Lamp();

// call methods turnOn() and turnOff()

l1.turnOn();

l2.turnOff();

}

}

**Output**:

Light on? true

Light on? false

In the above program,

1. We have created a class named Lamp.
2. The class has an instance variable isOn and two methods turnOn() and turnOff().
3. Inside the Main class, we have created two objects l1 and l2 of the Lamp class.
4. We then use the l1 object to call turnOn() and the l2 object to call turnOff():
5. l1.turnOn();

l2.turnOff();

1. The turnOn() method sets the isOn variable of l1 object to true. and prints the output. Similarly, the turnOff() method sets the isOn variable of the l2 object to false and prints the output.

**Note**: The variables defined inside a class are known as instance variables for a reason. When an object is created, it is called an instance of the class.

Each instance contains its own copy of the variables defined inside the class. Hence, known as instance variables. For example, the isOn variable is different for objects l1 and l2.

In the coming tutorials, we will be exploring more about object-oriented programming in Java.

## Java Array

An **array** is a dynamically-created object. It serves as a container that holds the constant number of values of the same type. It has a contiguous memory location. Once an array is created, we cannot change its size. We can create an array by using the following statement:

1. **int** array[]=**new** **int**[size];

The above statement creates an array of the specified size. When we try to add more than its size, it throws **ArrayIndexOutOfBoundsException**. For example:

1. **int** arr[]=**new** **int**[3];   //specified size of array is 3
2. //adding 4 elements into array
3. arr[0]=12;
4. arr[1]=2;
5. arr[2]=15;
6. arr[3]=67;

## Java ArrayList

The ArrayList class is a resizable [array](https://www.w3schools.com/java/java_arrays.asp), which can be found in the java.util package.

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want. The syntax is also slightly different:

### Example

Create an ArrayList object called **cars** that will store strings:

import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object

## Add Items

The ArrayList class has many useful methods. For example, to add elements to the ArrayList, use the add() method:

### Example

import java.util.ArrayList;

public class MyClass {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars);

}

}

Access an Item

To access an element in the ArrayList, use the get() method and refer to the index number:

### Example

cars.get(0);

## Change an Item

To modify an element, use the set() method and refer to the index number:

### Example

cars.set(0, "Opel");

## Remove an Item

To remove an element, use the remove() method and refer to the index number:

### Example

cars.remove(0);

To remove all the elements in the ArrayList, use the clear() method:

### Example

cars.clear();

**ArrayList Size**

To find out how many elements an ArrayList have, use the size method:

### Example

cars.size();

**Loop Through an ArrayList**

Loop through the elements of an ArrayList with a for loop, and use the size() method to specify how many times the loop should run:

### Example

public class MyClass {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

for (int i = 0; i < cars.size(); i++) {

System.out.println(cars.get(i));

}

}

}

You can also loop through an ArrayList with the **for-each** loop:

### Example

public class MyClass {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

for (String i : cars) {

System.out.println(i);

}

}

}

|  |  |  |
| --- | --- | --- |
| **Basis** | **Array** | **ArrayList** |
| **Definition** | An **array** is a dynamically-created object. It serves as a container that holds the constant number of values of the same type. It has a contiguous memory location. | The **ArrayList** is a class of Java **Collections** framework. It contains popular classes like **Vector, HashTable**, and **HashMap**. |
| **Static/ Dynamic** | Array is **static** in size. | ArrayList is **dynamic** in size. |
| **Resizable** | An array is a **fixed-length** data structure. | ArrayList is a **variable-length** data structure. It can be resized itself when needed. |
| **Initialization** | It is mandatory to provide the size of an array while initializing it directly or indirectly. | We can create an instance of ArrayList without specifying its size. Java creates ArrayList of default size. |
| **Performance** | It performs **fast** in comparison to ArrayList because of fixed size. | ArrayList is internally backed by the array in Java. The resize operation in ArrayList slows down the performance. |
| **Iterating Values** | We use **for** loop or **for each** loop to iterate over an array. | We use an **iterator** to iterate over ArrayList. |
| **Type-Safety** | We cannot use generics along with array because it is not a convertible type of array. | ArrayList allows us to store only **generic/ type, that's why it is type-safe.** |
| **Length** | Array provides a **length** variable which denotes the length of an array. | ArrayList provides the **size()** method to determine the size of ArrayList. |
| **Adding Elements** | We can add elements in an array by using the **assignment**operator. | Java provides the **add()** method to add elements in the ArrayList. |
| **Single/ Multi-Dimensional** | Array can be **multi-dimensional**. | ArrayList is always **single-dimensional**. |

## Example of Array in Java

In the following example, we have simply created an array of length four.

**public** **class** ArrayExample

{

**public** **static** **void** main(String args[])

{

//creating an array of integer type

**int** arr[]=**new** **int**[4];

//adding elements into array

arr[0]=12;

arr[1]=2;

arr[2]=15;

arr[3]=67;

**for**(**int** i=0;i<arr.length;i++)

{

System.out.println(arr[i]);

}

}

}

**Output**

12

2

15

67

**Example of ArrayList in Java**

In the following example, we have created an instance of ArrayList and performing iteration over the ArrayList.

**import** java.util.\*;

**public** **class** ArrayListExample

{

**public** **static** **void** main(String args[])

{

//creating an instance of ArrayList

List<Float> list = **new** ArrayList<Float>();

//adding element to arraylist

list.add(12.4f);

list.add(34.6f);

list.add(56.8f);

list.add(78.9f);

//iteration over ArrayList using for-each loop

**for**(Float f:list)

{

System.out.println(f);

}

}

} **Output** **12.4**

**34.6**

**56.8**

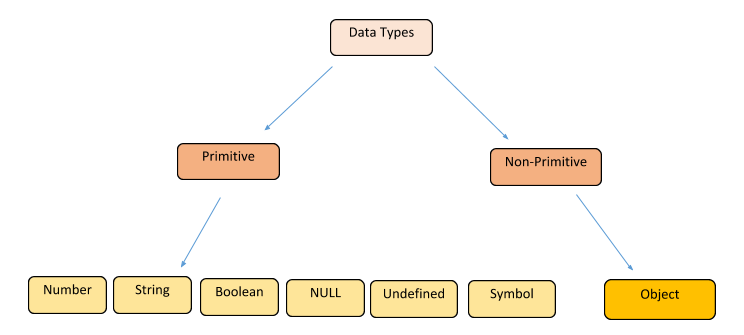
**78.9**

***Unit –Java Script***

**Data Types in Java Script :**

JavaScript is a scripting language for the web. Like any other programming language, it has its own data types. A data type in a language defines the type of data a variable can hold. In this article, we will discuss the various data types in JavaScript in detail:

JavaScript has seven types. Types are values that JavaScript can have. Below is a list of data types that JavaScript can have:

1. Number
2. String
3. Boolean
4. Undefined
5. Null
6. Object
7. Symbol

The ‘Symbol’ data type is new in JavaScript. It has been included in the ES6 version. We can find the type of value or data by using the ‘typeof’ JavaScript operator. The above data types in JavaScript are divided into two broad categories, primitive and non-primitive.

* The Primitive Data types in JavaScript include Number, String, Boolean, Undefined, Null and Symbol.
* The Non-Primitive data type has only one member i.e. the Object.

## ****Difference Between Primitive and Non-Primitive Data Types in JavaScript****

JavaScript primitive data types are data types that refer to a single value.

**E.g. var a =5;**

Here the variable ‘a’ is an integer data type and has a single integer value. The variable ‘a’ refers to a single value in memory. If we want to change the value of a, we would have to assign a new value to a.  *Primitive data types are not mutable.*

When we create a variable, it reserves a space for itself in the memory. The variable ‘a’ has space in memory which holds its value. When we try to change the value of ‘a’ by assigning another value like var a = 6, it doesn’t alter the value of the original a, it just creates a new variable ‘a’ with the new value 6.

We can assign the value of ‘a’ to another variable like this:

**var a1=a;**

Here the variable ‘a1’ is assigned the value of ‘a’, not the address of ‘a’ in memory.

Thus ‘a1’ now holds the same value as ‘a’.

We can compare the two variables ‘a’ and ‘a1’ as the two variables refer to the same value now.

Using the comparison operator will return a Boolean value of ‘true’.

**a===a1 // equals to ‘true’ as ‘===’** checks both the value and type of these two variables are true.

JavaScript non-primitive types are objects. An object holds a reference/address of a single key-value pair or many key-value pairs. Whenever we refer to an object, we refer to an address in memory which contains the key-value pair. If we assign an object ‘object1’ to another object ‘object2’, we are actually assigning the address of ‘object1’ to ‘object2’ instead of the key-value pair which the ‘object1’ contains in memory. Let’s see below”.

**var object1= {a:5, a1:6};**

**var object2 = object1;**

The above statement assigns the address of object2 to object1, and not the value {a:5, a1:6}. Thus with this assignment ‘object1’ and ‘object2’ refer to the same address in memory.

When we compare these two objects, we find that both of them refer to the same address in memory.

## ****Primitive Data Types****

Primitive data types are number, string, boolean, NULL, Infinity and symbol. Non-primitive data types is the object. The JavaScript arrays and functions are also objects.

**Numbers:**

A number data type can be an integer, a floating point value, an exponential value, a ‘NaN’ or a ‘Infinity’.

var a=250;  // integer value

var b=25.5;  // a number containing a decimal

var c = 10e4 //  an exponential value which evaluates to 10\*10000;

|  |  |
| --- | --- |
|  |  |

There are special numeric values e.g. NaN and Infinity.

If a number is divided by 0, the resulting value is infinity.

5/0;    // results in infinity

The type of infinity is a number

typeof(infinity);   // returns number

A ‘NaN’ results when we try to perform an operation on a number with a non-numeric value

‘hi’ \* 5; // returns NaN

typeof(NaN);  // returns a number

|  |  |
| --- | --- |
|  |  |

 We can also create a number literal by using the Number() function:

var c = Number(5);

console.log(c);  // This will return 5

We can create a number object using the ‘new’ operator and the Number() constructor:

var num1= new Number(5);

console.log(num1); // This will return 5

typeof(num1); // This will return ‘number’

**String:**

The string data type in JavaScript can be any group of characters enclosed by a single or double-quotes or by backticks.

var str1 = “This is a string1”;  // This is a string primitive type or string literal

var str2= ‘This is a string2’;

var str3 = `This is a string3`;

**Boolean:**

The boolean data type has only two values, true and false. It is mostly used to check a logical condition. Thus Booleans are logical data types which can be used for comparison of two variables or to check a condition. The true and false implies a ‘yes’ for ‘true’ and a ‘no’ for ‘false in some places when we check a condition or the existence of a variable or a value.

When we check the data type of ‘true’ or ‘false’ using typeof operator, it returns a boolean.

typeof(true) // returns boolean

typeof(false) // returns Boolean

Let’s see an example of comparison statement:

var a =5;

var b=6;

a==b // returns false

**Undefined:**

Undefined data type means a variable that is not defined. The variable is declared but doesn’t contain any value.

var a;

console.log(a); // This will return undefined

|  |
| --- |
|  |

The variable ‘a’ has been declared but hasn’t been assigned a value yet.  
We can assign a value to a:

a=5;

console.log(a); // This will return 5

**Null:**

The null in JavaScript is a data type that is represented by only one value, the ‘null’ itself. A null value means no value.

var a = null;

console.log(a);   // This returns null

**Symbol:**

The ‘symbol’ data type is new in es6. It is one of the new features of es6. The symbol data type defines a property of an object which is private to the object. It refers to the ‘key’ of the key-value pair of an object.

## ****The Non-Primitive Data Types****

The ‘object’ is a non-primitive data type in JavaScript. Arrays and Functions in JavaScript belong to the ‘object’ data type.

**Object:**

Let’s create an object literal. An object in JavaScript contains key-value pairs in its address. When we refer to obj1, we are actually referring to the address in memory which contains the value {a: 5, b: 6}, instead of the value {a: 5, b: 6} directly.

var obj1 = { a: 5, b: 6 };

We can change or mutate the value of obj1.

obj1[a] =7;

console.log(obj1) // will return the value {a: 7, b: 6}

Thus the value has changed successfully.

**JavaScript Operators**

JavaScript includes operators as in other languages. An operator performs some operation on single or multiple operands (data value) and produces a result. For example 1 + 2, where + sign is an operator and 1 is left operand and 2 is right operand. + operator adds two numeric values and produces a result which is 3 in this case.

Syntax:

<*Left operand*> operator <*right operand*>

<*Left operand*> operator

JavaScript includes following categories of operators.

1. Arithmetic Operators
2. Comparison Operators
3. Logical Operators
4. Assignment Operators
5. Conditional Operators

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations between numeric operands.

| **Operator** | **Description** |
| --- | --- |
| + | Adds two numeric operands. |
| - | Subtract right operand from left operand |
| \* | Multiply two numeric operands. |
| / | Divide left operand by right operand. |
| % | Modulus operator. Returns remainder of two operands. |
| ++ | Increment operator. Increase operand value by one. |
| -- | Decrement operator. Decrease value by one. |

The following example demonstrates how arithmetic operators perform different tasks on operands. Ex:-

var x = 5, y = 10, z = 15;

x + y; //returns 15 y / x; //returns 2

y - x; //returns 5 x \* y; //returns 50

x % 2; //returns 1 x++; //returns 6

x--; //returns 4

**+** operator performs concatenation operation when one of the operands is of string type.

The following example shows how

**+** operator performs operation on operands of different data types.

var a = 5, b = "Hello ", c = "World!", d = 10;

a + b; // "5Hello "

b + c; // "Hello World!"

a + d; // 15

## Comparison Operators

JavaScript language includes operators that compare two operands and return Boolean value true or false.

| Operators | Description |
| --- | --- |
| == | Compares the equality of two operands without considering type. |
| === | Compares equality of two operands with type. |
| != | Compares inequality of two operands. |
| > | Checks whether left side value is greater than right side value. If yes then returns true otherwise false. |
| < | Checks whether left operand is less than right operand. If yes then returns true otherwise false. |
| >= | Checks whether left operand is greater than or equal to right operand. If yes then returns true otherwise false. |
| <= | Checks whether left operand is less than or equal to right operand. If yes then returns true otherwise false. |

The following example demonstrates how comparison operators perform different tasks.

var a = 5, b = 10, c = "5";

var x = a;

a == c; // returns true

a === c; // returns false

a == x; // returns true

a != b; // returns true

a > b; // returns false

a < b; // returns true

a >= b; // returns false

a <= b; // returns true

a >= c; // returns true

a <= c; // returns true

## Logical Operators

Logical operators are used to combine two or more conditions. JavaScript includes following logical operators.

| Operator | Description |
| --- | --- |
| && | && is known as AND operator. It checks whether two operands are non-zero (0, false, undefined, null or "" are considered as zero), if yes then returns 1 otherwise 0. |
| || | || is known as OR operator. It checks whether any one of the two operands is non-zero (0, false, undefined, null or "" is considered as zero). |
| ! | ! is known as NOT operator. It reverses the boolean result of the operand (or condition) |

Ex:

var a = 5, b = 10;

(a != b) && (a < b); // returns true

(a > b) || (a == b); // returns false

(a < b) || (a == b); // returns true

!(a < b); // returns false

!(a > b); // returns true

## Assignment Operators

JavaScript includes assignment operators to assign values to variables with less key strokes.

| Assignment operators | Description |
| --- | --- |
| = | Assigns right operand value to left operand. |
| += | Sums up left and right operand values and assign the result to the left operand. |
| -= | Subtract right operand value from left operand value and assign the result to the left operand. |
| \*= | Multiply left and right operand values and assign the result to the left operand. |
| /= | Divide left operand value by right operand value and assign the result to the left operand. |
| %= | Get the modulus of left operand divide by right operand and assign resulted modulus to the left operand. |

Ex:

var x = 5, y = 10, z = 15;

x = y; //x would be 10

x += 1; //x would be 6

x -= 1; //x would be 4

x \*= 5; //x would be 25

x /= 5; //x would be 1

x %= 2; //x would be 1

## Ternary Operator

## JavaScript includes special operator called ternary operator :? that assigns a value to a variable based on some condition. This is like short form of if-else condition.

Syntax:

<condition> ? <value1> : <value2>;

Ternary operator starts with conditional expression followed by ? operator. Second part ( after ? and before : operator) will be executed if condition turns out to be true. If condition becomes false then third part (after :) will be executed.

Example :

var a = 10, b = 5;

var c = a > b? a : b; // value of c would be 10

var d = a > b? b : a; // value of d would be 5

## Control Structure

Control structure actually controls the flow of execution of a program. Following are the several control structure supported by javascript.

* if … else
* switch case
* do while loop
* while loop
* for loop

### If … else

The if statement is the fundamental control statement that allows JavaScript to make decisions and execute statements conditionally.

Syntax

if (expression){

Statement(s) to be executed if expression is true

}

Example

<script type="text/javascript">

<!--

var age = 20;

if( age > 18 ){

document.write("<b>Qualifies for driving</b>");

}

//-->

</script>

### Switch case

The basic syntax of the switch statement is to give an expression to evaluate and several different statements to execute based on the value of the expression. The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

Syntax

switch (expression) {

case condition 1: statement(s)

break;

case condition 2: statement(s)

break;

...

case condition n: statement(s)

break;

default: statement(s)

}

Example

<script type="text/javascript">

<!--

var grade='A';

document.write("Entering switch block<br/>");

switch (grade) {

case 'A': document.write("Good job<br/>");

break;

case 'B': document.write("Pretty good<br/>");

break;

case 'C': document.write("Passed<br/>");

break;

case 'D': document.write("Not so good<br/>");

break;

case 'F': document.write("Failed<br/>");

break;

default: document.write("Unknown grade<br/>")

}

document.write("Exiting switch block");

//-->

</script>

### Do while Loop

The do...while loop is similar to the while loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once, even if the condition is false.

Syntax

do{

Statement(s) to be executed;

} while (expression);

Example

<script type="text/javascript">

<!--

var count = 0;

document.write("Starting Loop" + "<br/>");

do{

document.write("Current Count : " + count + "<br/>");

count++;

}while (count < 0);

document.write("Loop stopped!");

//-->

</script>

This will produce following result −

Starting Loop

Current Count : 0

Loop stopped! While Loop

The purpose of a while loop is to execute a statement or code block repeatedly as long as expression is true. Once expression becomes false, the loop will be exited.

Syntax

while (expression){

Statement(s) to be executed if expression is true

}

Example

<script type="text/javascript">

<!--

var count = 0;

document.write("Starting Loop" + "<br/>");

while (count < 10){

document.write("Current Count : " + count + "<br/>");

count++;

}

document.write("Loop stopped!");

//-->

</script>

This will produce following result −

Starting Loop

Current Count : 0

Current Count : 1

Current Count : 2

Current Count : 3

Current Count : 4

Current Count : 5

Current Count : 6

Current Count : 7

Current Count : 8

Current Count : 9

Loop stopped!

### For Loop

The for loop is the most compact form of looping and includes the following three important parts −

* The loop initialization where we initialize our counter to a starting value. The initialization statement is executed before the loop begins.
* The test statement which will test if the given condition is true or not. If condition is true then code given inside the loop will be executed otherwise loop will come out.
* The iteration statement where you can increase or decrease your counter.

Syntax

for (initialization; test condition; iteration statement){

Statement(s) to be executed if test condition is true

}

Example

<script type="text/javascript">

<!--

var count;

document.write("Starting Loop" + "<br/>");

for(count = 0; count < 10; count++){

document.write("Current Count : " + count );

document.write("<br/>");

}

document.write("Loop stopped!");

//-->

</script>

This will produce following result which is similar to while loop −

Starting Loop

Current Count : 0

Current Count : 1

Current Count : 2

Current Count : 3

Current Count : 4

Current Count : 5

Current Count : 6

Current Count : 7

Current Count : 8

Current Count : 9

Loop stopped!

## Creating Sample Program

Following is the sample program that shows time, when we click in button.

<html>

<body>

<button onclick="this.innerHTML=Date()">The time is?</button>

<p>Click to display the date.</p>

<button onclick="displayDate()">The time is?</button>

<script>

function displayDate() {

document.getElementById("demo").innerHTML = Date();

}</script>

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

</script>

</body>

</html>

**Output**

**JavaScript - Events**

## What is an Event?

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page.

When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc.

Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable.

Events are a part of the Document Object Model (DOM) Level 3 and every HTML element contains a set of events which can trigger JavaScript Code.

Please go through this small tutorial for a better understanding [HTML Event Reference](https://www.tutorialspoint.com/html/html_events_ref.htm). Here we will see a few examples to understand a relation between Event and JavaScript −

## onclick Event Type

This is the most frequently used event type which occurs when a user clicks the left button of his mouse. You can put your validation, warning etc., against this event type.

### Example

Try the following example.

<html>

<head>

<script type = "text/javascript">

<!--

function sayHello() {

alert("Hello World")

}

//-->

</script>

</head>

<body>

<p>Click the following button and see result</p>

<form>

<input type = "button" onclick = "sayHello()" value = "Say Hello" />

</form>

</body>

</html>

Output:-

Click the following button and see result

Top of Form

Bottom of Form

Say Hello

## onsubmit Event Type

**onsubmit** is an event that occurs when you try to submit a form. You can put your form validation against this event type.

### Example

The following example shows how to use onsubmit. Here we are calling a **validate()** function before submitting a form data to the webserver. If **validate()** function returns true, the form will be submitted, otherwise it will not submit the data.

Try the following example.

<html>

<head>

<script type = "text/javascript">

<!--

function validation() {

all validation goes here

.........

return either true or false

}

//-->

</script>

</head>

<body>

<form method = "POST" action = "t.cgi" onsubmit = "return validate()">

.......

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

</form>

</body>

</html>

## nmouseover and onmouseout

These two event types will help you create nice effects with images or even with text as well. The **onmouseover** event triggers when you bring your mouse over any element and the **onmouseout** triggers when you move your mouse out from that element. Try the following example.

<html>

<head>

<script type = "text/javascript">

<!--

function over() {

document.write ("Mouse Over");

}

function out() {

document.write ("Mouse Out");

}

//-->

</script>

</head>

<body>

<p>Bring your mouse inside the division to see the result:</p>

<div onmouseover = "over()" onmouseout = "out()">

<h2> This is inside the division </h2>

</div>

</body>

</html>

### Output

Mouse Over

**UNIT- JDBC**

## What is JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, 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 mentioned below that are commonly associated with database usage.

* Making a connection to a database.
* Creating SQL or MySQL statements.
* Executing 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 ServerPages (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.

## Pre-Requisite

Before moving further, you need to have a good understanding of the following two subjects −

* [Core JAVA Programming](https://www.tutorialspoint.com/java/index.htm)
* [SQL or MySQL Database](https://www.tutorialspoint.com/mysql/index.htm)

## JDBC Architecture

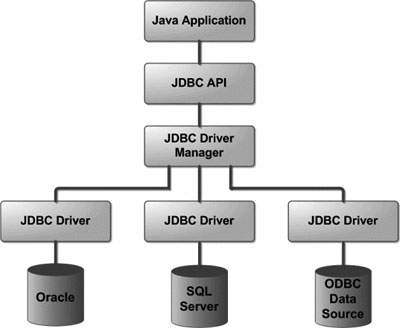
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

* **JDBC API:** This provides the application-to-JDBC Manager connection.
* **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

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 −

* **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 subprotocol under JDBC will be used to establish a database Connection.
* **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 manages objects of this type. It also abstracts the details associated with working with Driver objects.
* **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
* **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
* **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.
* **SQLException:** This class handles any errors that occur in a database application.

## The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas −

* Automatic database driver loading.
* Exception handling improvements.
* Enhanced BLOB/CLOB functionality.
* Connection and statement interface enhancements.
* National character set support.
* SQL ROWID access.
* SQL 2003 XML data type support.
* Annotations.
* **S**tructured **Q**uery **L**anguage (SQL) is a standardized language that allows you to perform operations on a database, such as creating entries, reading content, updating content, and deleting entries.
* SQL is supported by almost any database you will likely use, and it allows you to write database code independently of the underlying database.
* This chapter gives an overview of SQL, which is a prerequisite to understand JDBC concepts. After going through this chapter, you will be able to Create, **C**reate, **R**ead, **U**pdate, and **D**elete (often referred to as **CRUD** operations) data from a database.
* For a detailed understanding on SQL, you can read our [MySQL Tutorial](https://www.tutorialspoint.com/mysql/index.htm).

## Create Database

* The CREATE DATABASE statement is used for creating a new database. The syntax is −
* SQL> CREATE DATABASE DATABASE\_NAME;

## Example

* The following SQL statement creates a Database named EMP −
* SQL> CREATE DATABASE EMP;

## Drop Database

* The DROP DATABASE statement is used for deleting an existing database. The syntax is −
* SQL> DROP DATABASE DATABASE\_NAME;
* **Note:** To create or drop a database you should have administrator privilege on your database server. Be careful, deleting a database would loss all the data stored in the database.

## Create Table

* The CREATE TABLE statement is used for creating a new table. The syntax is −
* SQL> CREATE TABLE table\_name
* (
* column\_name column\_data\_type,
* column\_name column\_data\_type,
* column\_name column\_data\_type
* ...
* );

## Example

* The following SQL statement creates a table named Employees with four columns −
* SQL> CREATE TABLE Employees
* (
* id INT NOT NULL,
* age INT NOT NULL,
* first VARCHAR(255),
* last VARCHAR(255),
* PRIMARY KEY ( id )
* );

## Drop Table

* The DROP TABLE statement is used for deleting an existing table. The syntax is −
* SQL> DROP TABLE table\_name;

## Example

* The following SQL statement deletes a table named Employees −
* SQL> DROP TABLE Employees;

## INSERT Data

* The syntax for INSERT, looks similar to the following, where column1, column2, and so on represents the new data to appear in the respective columns −
* SQL> INSERT INTO table\_name VALUES (column1, column2, ...);

## Example

* The following SQL INSERT statement inserts a new row in the Employees database created earlier −
* SQL> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');

## SELECT Data

* The SELECT statement is used to retrieve data from a database. The syntax for SELECT is −
* SQL> SELECT column\_name, column\_name, ...
* FROM table\_name
* WHERE conditions;
* The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

## Example

* The following SQL statement selects the age, first and last columns from the Employees table, where id column is 100 −
* SQL> SELECT first, last, age
* FROM Employees
* WHERE id = 100;
* The following SQL statement selects the age, first and last columns from the Employees table where *first* column contains *Zara* −
* SQL> SELECT first, last, age
* FROM Employees
* WHERE first LIKE '%Zara%';

## UPDATE Data

* The UPDATE statement is used to update data. The syntax for UPDATE is −
* SQL> UPDATE table\_name
* SET column\_name = value, column\_name = value, ...
* WHERE conditions;
* The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

## Example

* The following SQL UPDATE statement changes the age column of the employee whose id is 100 −
* SQL> UPDATE Employees SET age=20 WHERE id=100;

## DELETE Data

* The DELETE statement is used to delete data from tables. The syntax for DELETE is −
* SQL> DELETE FROM table\_name WHERE conditions;
* The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

## Example

* The following SQL DELETE statement deletes the record of the employee whose id is 100 −
* SQL> DELETE FROM Employees WHERE id=100;

# JDBC - Environment Setup

To start developing with JDBC, you should setup your JDBC environment by following the steps shown below. We assume that you are working on a Windows platform.

## Install Java

Install J2SE Development Kit 5.0 (JDK 5.0) from [Java Official Site](http://java.sun.com/j2se/1.5.0/download.jsp).

Make sure following environment variables are set as described below −

* **JAVA\_HOME:** This environment variable should point to the directory where you installed the JDK, e.g. C:\Program Files\Java\jdk1.5.0.
* **CLASSPATH:** This environment variable should have appropriate paths set, e.g. C:\Program Files\Java\jdk1.5.0\_20\jre\lib.
* **PATH:** This environment variable should point to appropriate JRE bin, e.g. C:\Program Files\Java\jre1.5.0\_20\bin.

It is possible you have these variable set already, but just to make sure here's how to check.

* Go to the control panel and double-click on System. If you are a Windows XP user, it is possible you have to open Performance and Maintenance before you will see the System icon.
* Go to the Advanced tab and click on the Environment Variables.
* Now check if all the above mentioned variables are set properly.

You automatically get both JDBC packages **java.sql** and **javax.sql,** when you install J2SE Development Kit 5.0 (JDK 5.0).

## Install Database

The most important thing you will need, of course is an actual running database with a table that you can query and modify.

Install a database that is most suitable for you. You can have plenty of choices and most common are −

* **MySQL DB:** MySQL is an open source database. You can download it from [MySQL Official Site](http://dev.mysql.com/downloads/mysql). We recommend downloading the full Windows installation.

In addition, download and install [MySQL Administrator](http://dev.mysql.com/downloads/gui-tools/) as well as [MySQL Query Browser.](http://dev.mysql.com/downloads/gui-tools/) These are GUI based tools that will make your development much easier.

Finally, download and unzip [MySQL Connector/J](http://dev.mysql.com/downloads/connector/j/3.1.html) (the MySQL JDBC driver) in a convenient directory. For the purpose of this tutorial we will assume that you have installed the driver at C:\Program Files\MySQL\mysql-connector-java-5.1.8.

Accordingly, set CLASSPATH variable to C:\Program Files\MySQL\mysql-connector-java-5.1.8\mysql-connector-java-5.1.8-bin.jar. Your driver version may vary based on your installation.

* **PostgreSQL DB:** PostgreSQL is an open source database. You can download it from [PostgreSQL Official Site](http://www.postgresql.org/download/).

The Postgres installation contains a GUI based administrative tool called pgAdmin III. JDBC drivers are also included as part of the installation.

* **Oracle DB:** Oracle DB is a commercial database sold by Oracle . We assume that you have the necessary distribution media to install it.

Oracle installation includes a GUI based administrative tool called Enterprise Manager. JDBC drivers are also included as a part of the installation.

## Install Database Drivers

The latest JDK includes a JDBC-ODBC Bridge driver that makes most Open Database Connectivity (ODBC) drivers available to programmers using the JDBC API.

Now a days, most of the Database vendors are supplying appropriate JDBC drivers along with Database installation. So, you should not worry about this part.

## Set Database Credential

For this tutorial we are going to use MySQL database. When you install any of the above database, its administrator ID is set to **root** and gives provision to set a password of your choice.

Using root ID and password you can either create another user ID and password, or you can use root ID and password for your JDBC application.

There are various database operations like database creation and deletion, which would need administrator ID and password.

For rest of the JDBC tutorial, we would use MySQL Database with **username** as ID and **password** as password.

If you do not have sufficient privilege to create new users, then you can ask your Database Administrator (DBA) to create a user ID and password for you.

## Create Database

To create the **EMP** database, use the following steps −

## Step 1

Open a **Command Prompt** and change to the installation directory as follows −

C:\>

C:\>cd Program Files\MySQL\bin

C:\Program Files\MySQL\bin>

**Note:** The path to **mysqld.exe** may vary depending on the install location of MySQL on your system. You can also check documentation on how to start and stop your database server.

## Step 2

Start the database server by executing the following command, if it is already not running.

C:\Program Files\MySQL\bin>mysqld

C:\Program Files\MySQL\bin>

## Step 3

Create the **EMP** database by executing the following command −

C:\Program Files\MySQL\bin> mysqladmin create EMP -u root -p

Enter password: \*\*\*\*\*\*\*\*

C:\Program Files\MySQL\bin>

## Create Table

To create the **Employees** table in EMP database, use the following steps −

## Step 1

Open a **Command Prompt** and change to the installation directory as follows −

C:\>

C:\>cd Program Files\MySQL\bin

C:\Program Files\MySQL\bin>

## Step 2

Login to the database as follows −

C:\Program Files\MySQL\bin>mysql -u root -p

Enter password: \*\*\*\*\*\*\*\*

mysql>

## Step 3

Create the table **Employee** as follows −

mysql> use EMP;

mysql> create table Employees

-> (

-> id int not null,

-> age int not null,

-> first varchar (255),

-> last varchar (255)

-> );

Query OK, 0 rows affected (0.08 sec)

mysql>

## Create Data Records

Finally you create few records in Employee table as follows −

mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');

Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');

Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');

Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');

Query OK, 1 row affected (0.00 sec)

mysql>

For a complete understanding on MySQL database, study the [MySQL Tutorial](https://www.tutorialspoint.com/mysql/index.htm).

Now you are ready to start experimenting with JDBC. Next chapter gives you a sample example on JDBC Programming.

# JDBC - Sample, Example Code

This chapter provides an example of how to create a simple JDBC application. This will show you how to open a database connection, execute a SQL query, and display the results.

All the steps mentioned in this template example, would be explained in subsequent chapters of this tutorial.

## Creating JDBC Application

There are following six steps involved in building a JDBC application −

* **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.\** will suffice.
* **Register the JDBC driver:** Requires that you initialize a driver so you can open a communication channel with the database.
* **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
* **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
* **Extract data from result set:** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
* **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

## Sample Code

This sample example can serve as a **template** when you need to create your own JDBC application in the future.

This sample code has been written based on the environment and database setup done in the previous chapter.

Copy and paste the following example in FirstExample.java, compile and run as follows −

//STEP 1. Import required packages

import java.sql.\*;

public class FirstExample {

// JDBC driver name and database URL

static final String JDBC\_DRIVER = "com.mysql.jdbc.Driver";

static final String DB\_URL = "jdbc:mysql://localhost/EMP";

// Database credentials

static final String USER = "username";

static final String PASS = "password";

public static void main(String[] args) {

Connection conn = null;

Statement stmt = null;

try{

//STEP 2: Register JDBC driver

Class.forName("com.mysql.jdbc.Driver");

//STEP 3: Open a connection

System.out.println("Connecting to database...");

conn = DriverManager.getConnection(DB\_URL,USER,PASS);

//STEP 4: Execute a query

System.out.println("Creating statement...");

stmt = conn.createStatement();

String sql;

sql = "SELECT id, first, last, age FROM Employees";

ResultSet rs = stmt.executeQuery(sql);

//STEP 5: Extract data from result set

while(rs.next()){

//Retrieve by column name

int id = rs.getInt("id");

int age = rs.getInt("age");

String first = rs.getString("first");

String last = rs.getString("last");

//Display values

System.out.print("ID: " + id);

System.out.print(", Age: " + age);

System.out.print(", First: " + first);

System.out.println(", Last: " + last);

}

//STEP 6: Clean-up environment

rs.close();

stmt.close();

conn.close();

}catch(SQLException se){

//Handle errors for JDBC

se.printStackTrace();

}catch(Exception e){

//Handle errors for Class.forName

e.printStackTrace();

}finally{

//finally block used to close resources

try{

if(stmt!=null)

stmt.close();

}catch(SQLException se2){

}// nothing we can do

try{

if(conn!=null)

conn.close();

}catch(SQLException se){

se.printStackTrace();

}//end finally try

}//end try

System.out.println("Goodbye!");

}//end main

}//end FirstExample

Now let us compile the above example as follows −

C:\>javac FirstExample.java

C:\>

When you run **FirstExample**, it produces the following result −

C:\>java FirstExample

Connecting to database...

Creating statement...

ID: 100, Age: 18, First: Zara, Last: Ali

ID: 101, Age: 25, First: Mahnaz, Last: Fatma

ID: 102, Age: 30, First: Zaid, Last: Khan

ID: 103, Age: 28, First: Sumit, Last: Mittal

C:\>

**UNIT-Java Server Pages (JSP)**

Java Server Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications. JSP have access to the entire family of Java APIs, including the JDBC API to access enterprise databases. This tutorial will teach you how to use Java Server Pages to develop your web applications in simple and easy steps.

## Why to Learn JSP?

JavaServer Pages often serve the same purpose as programs implemented using the **Common Gateway Interface (CGI)**. But JSP offers several advantages in comparison with the CGI.

* Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself instead of having separate CGI files.
* JSP are always compiled before they are processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested.
* JavaServer Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including **JDBC, JNDI, EJB, JAXP,** etc.
* JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

Finally, JSP is an integral part of Java EE, a complete platform for enterprise class applications. This means that JSP can play a part in the simplest applications to the most complex and demanding.

## Applications of JSP

As mentioned before, JSP is one of the most widely used language over the web. I'm going to list few of them here:

### JSP vs. Active Server Pages (ASP)

The advantages of JSP are twofold. First, the dynamic part is written in Java, not Visual Basic or other MS specific language, so it is more powerful and easier to use. Second, it is portable to other operating systems and non-Microsoft Web servers.

### JSP vs. Pure Servlets

It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.

### JSP vs. Server-Side Includes (SSI)

SSI is really only intended for simple inclusions, not for "real" programs that use form data, make database connections, and the like.

### JSP vs. JavaScript

JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.

### JSP vs. Static HTML

Regular HTML, of course, cannot contain dynamic information.

## What is JavaServer Pages?

JavaServer Pages (JSP) is a technology for developing Webpages that supports dynamic content. This helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>.

A JavaServer Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.

Using JSP, you can collect input from users through Webpage forms, present records from a database or another source, and create Webpages dynamically.

JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages, and sharing information between requests, pages etc.

## Why Use JSP?

JavaServer Pages often serve the same purpose as programs implemented using the **Common Gateway Interface (CGI)**. But JSP offers several advantages in comparison with the CGI.

* Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself instead of having separate CGI files.
* JSP are always compiled before they are processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested.
* JavaServer Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including **JDBC, JNDI, EJB, JAXP,** etc.
* JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

Finally, JSP is an integral part of Java EE, a complete platform for enterprise class applications. This means that JSP can play a part in the simplest applications to the most complex and demanding.

## Advantages of JSP

Following table lists out the other advantages of using JSP over other technologies −

### vs. Active Server Pages (ASP)

The advantages of JSP are twofold. First, the dynamic part is written in Java, not Visual Basic or other MS specific language, so it is more powerful and easier to use. Second, it is portable to other operating systems and non-Microsoft Web servers.

### vs. Pure Servlets

It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.

### vs. Server-Side Includes (SSI)

SSI is really only intended for simple inclusions, not for "real" programs that use form data, make database connections, and the like.

### vs. JavaScript

JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.

# JSP - Environment Setup

A development environment is where you would develop your JSP programs, test them and finally run them.

This tutorial will guide you to setup your JSP development environment which involves the following steps −

## Setting up Java Development Kit

This step involves downloading an implementation of the Java Software Development Kit (SDK) and setting up the PATH environment variable appropriately.

You can download SDK from Oracle's Java site − [Java SE Downloads](https://www.oracle.com/technetwork/java/javase/downloads/index.html).

Once you download your Java implementation, follow the given instructions to install and configure the setup. Finally set the **PATH and JAVA\_HOME** environment variables to refer to the directory that contains **java** and **javac**, typically **java\_install\_dir/bin** and **java\_install\_dir** respectively.

* + If you are running Windows and install the SDK in **C:\jdk1.5.0\_20**, you need to add the following line in your **C:\autoexec.bat** file.

set PATH = C:\jdk1.5.0\_20\bin;%PATH%

set JAVA\_HOME = C:\jdk1.5.0\_20

Alternatively, on **Windows NT/2000/XP**, you can also right-click on **My Computer**, select **Properties**, then **Advanced**, followed by **Environment Variables**. Then, you would update the PATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if the SDK is installed in **/usr/local/jdk1.5.0\_20** and you use the C shell, you will put the following into your **.cshrc** file.

setenv PATH /usr/local/jdk1.5.0\_20/bin:$PATH

setenv JAVA\_HOME /usr/local/jdk1.5.0\_20

Alternatively, if you use an **Integrated Development Environment (IDE)** like **Borland JBuilder, Eclipse, IntelliJ IDEA**, or **Sun ONE Studio**, compile and run a simple program to confirm that the IDE knows where you installed Java.

## Setting up Web Server: Tomcat

A number of Web Servers that support JavaServer Pages and Servlets development are available in the market. Some web servers can be downloaded for free and Tomcat is one of them.

Apache Tomcat is an open source software implementation of the JavaServer Pages and Servlet technologies and can act as a standalone server for testing JSP and Servlets, and can be integrated with the Apache Web Server. Here are the steps to set up Tomcat on your machine −

* Download the latest version of Tomcat from <https://tomcat.apache.org/>.
* Once you downloaded the installation, unpack the binary distribution into a convenient location. For example, in **C:\apache-tomcat-5.5.29 on windows, or /usr/local/apache-tomcat-5.5.29** on Linux/Unix and create **CATALINA\_HOME** environment variable pointing to these locations.

Tomcat can be started by executing the following commands on the Windows machine −

%CATALINA\_HOME%\bin\startup.bat

or

C:\apache-tomcat-5.5.29\bin\startup.bat

Tomcat can be started by executing the following commands on the Unix (Solaris, Linux, etc.) machine −

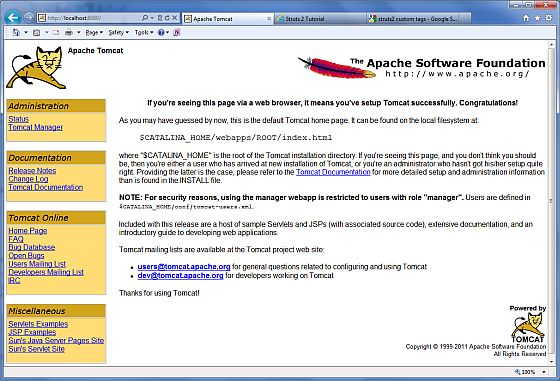
$CATALINA\_HOME/bin/startup.sh

or

/usr/local/apache-tomcat-5.5.29/bin/startup.sh

After a successful startup, the default web-applications included with Tomcat will be available by visiting **http://localhost:8080/**.

Upon execution, you will receive the following output −



Further information about configuring and running Tomcat can be found in the documentation included here, as well as on the Tomcat web site − <https://tomcat.apache.org/>.

Tomcat can be stopped by executing the following commands on the Windows machine −

%CATALINA\_HOME%\bin\shutdown

or

C:\apache-tomcat-5.5.29\bin\shutdown

Tomcat can be stopped by executing the following commands on Unix (Solaris, Linux, etc.) machine −

$CATALINA\_HOME/bin/shutdown.sh

or

/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh

## Setting up CLASSPATH

Since servlets are not part of the Java Platform, Standard Edition, you must identify the servlet classes to the compiler.

If you are running Windows, you need to put the following lines in your **C:\autoexec.bat** file.

set CATALINA = C:\apache-tomcat-5.5.29

set CLASSPATH = %CATALINA%\common\lib\jsp-api.jar;%CLASSPATH%

Alternatively, on **Windows NT/2000/XP**, you can also right-click on **My Computer**, select **Properties**, then **Advanced**, then **Environment Variables**. Then, you would update the CLASSPATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if you are using the C shell, you would put the following lines into your **.cshrc** file.

setenv CATALINA = /usr/local/apache-tomcat-5.5.29

setenv CLASSPATH $CATALINA/common/lib/jsp-api.jar:$CLASSPATH

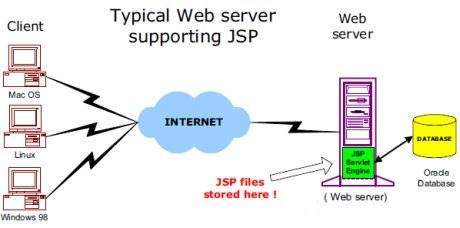
**NOTE** − Assuming that your development directory is **C:\JSPDev (Windows)** or **/usr/JSPDev (Unix)**, then you would need to add these directories as well in CLASSPATH.

The web server needs a JSP engine, i.e, a container to process JSP pages. The JSP container is responsible for intercepting requests for JSP pages. This tutorial makes use of Apache which has built-in JSP container to support JSP pages development.

# JSP - Architecture

A JSP container works with the Web server to provide the runtime environment and other services a JSP needs. It knows how to understand the special elements that are part of JSPs.

Following diagram shows the position of JSP container and JSP files in a Web application.

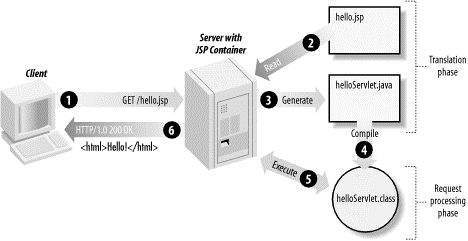


## JSP Processing

The following steps explain how the web server creates the Webpage using JSP −

* As with a normal page, your browser sends an HTTP request to the web server.
* The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with **.jsp** instead of **.html**.
* The JSP engine loads the JSP page from disk and converts it into a servlet content. This conversion is very simple in which all template text is converted to println( ) statements and all JSP elements are converted to Java code. This code implements the corresponding dynamic behavior of the page.
* The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.
* A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format. The output is furthur passed on to the web server by the servlet engine inside an HTTP response.
* The web server forwards the HTTP response to your browser in terms of static HTML content.
* Finally, the web browser handles the dynamically-generated HTML page inside the HTTP response exactly as if it were a static page.

All the above mentioned steps can be seen in the following diagram −



Typically, the JSP engine checks to see whether a servlet for a JSP file already exists and whether the modification date on the JSP is older than the servlet. If the JSP is older than its generated servlet, the JSP container assumes that the JSP hasn't changed and that the generated servlet still matches the JSP's contents. This makes the process more efficient than with the other scripting languages (such as PHP) and therefore faster.

So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming wiz. Except for the translation phase, a JSP page is handled exactly like a regular servlet.

# JSP - Implicit Objects

In this chapter, we will discuss the Implicit Objects in JSP. These Objects are the Java objects that the JSP Container makes available to the developers in each page and the developer can call them directly without being explicitly declared. JSP Implicit Objects are also called **pre-defined variables**.

Following table lists out the nine Implicit Objects that JSP supports −

|  |  |
| --- | --- |
| **S.No.** | **Object & Description** |
| 1 | **request**  This is the **HttpServletRequest** object associated with the request. |
| 2 | **response**  This is the **HttpServletResponse** object associated with the response to the client. |
| 3 | **out**  This is the **PrintWriter** object used to send output to the client. |
| 4 | **session**  This is the **HttpSession** object associated with the request. |
| 5 | **application**  This is the **ServletContext** object associated with the application context. |
| 6 | **config**  This is the **ServletConfig** object associated with the page. |
| 7 | **pageContext**  This encapsulates use of server-specific features like higher performance **JspWriters**. |
| 8 | **page**  This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class. |
| 9 | **Exception**  The **Exception** object allows the exception data to be accessed by designated JSP. |

## The request Object

The request object is an instance of a **javax.servlet.http.HttpServletRequest** object. Each time a client requests a page the JSP engine creates a new object to represent that request.

The request object provides methods to get the HTTP header information including form data, cookies, HTTP methods etc.

We can cover a complete set of methods associated with the request object in a subsequent chapter − [JSP - Client Request](https://www.tutorialspoint.com/jsp/jsp_client_request.htm).

## The response Object

The response object is an instance of a **javax.servlet.http.HttpServletResponse** object. Just as the server creates the request object, it also creates an object to represent the response to the client.

The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes, etc.

We will cover a complete set of methods associated with the response object in a subsequent chapter − [JSP - Server Response](https://www.tutorialspoint.com/jsp/jsp_server_response.htm).

## The out Object

The out implicit object is an instance of a **javax.servlet.jsp.JspWriter** object and is used to send content in a response.

The initial JspWriter object is instantiated differently depending on whether the page is buffered or not. Buffering can be easily turned off by using the **buffered = 'false'** attribute of the page directive.

The JspWriter object contains most of the same methods as the **java.io.PrintWriter** class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws **IOExceptions**.

Following table lists out the important methods that we will use to write **boolean char, int, double, object, String**, etc.

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1 | **out.print(dataType dt)**  Print a data type value |
| 2 | **out.println(dataType dt)**  Print a data type value then terminate the line with new line character. |
| 3 | **out.flush()**  Flush the stream. |

## The session Object

The session object is an instance of **javax.servlet.http.HttpSession** and behaves exactly the same way that session objects behave under Java Servlets.

The session object is used to track client session between client requests. We will cover the complete usage of session object in a subsequent chapter − [JSP - Session Tracking](https://www.tutorialspoint.com/jsp/jsp_session_tracking.htm).

## The application Object

The application object is direct wrapper around the **ServletContext** object for the generated Servlet and in reality an instance of a **javax.servlet.ServletContext** object.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the **jspDestroy()** method.

By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it.

We will check the use of Application Object in [JSP - Hits Counter](https://www.tutorialspoint.com/jsp/jsp_hits_counter.htm) chapter.

## The config Object

The config object is an instantiation of **javax.servlet.ServletConfig** and is a direct wrapper around the **ServletConfig** object for the generated servlet.

This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc.

The following **config** method is the only one you might ever use, and its usage is trivial −

config.getServletName();

This returns the servlet name, which is the string contained in the **<servlet-name>** element defined in the **WEB-INF\web.xml** file.

## The pageContext Object

The pageContext object is an instance of a **javax.servlet.jsp.PageContext** object. The pageContext object is used to represent the entire JSP page.

This object is intended as a means to access information about the page while avoiding most of the implementation details.

This object stores references to the request and response objects for each request. The **application, config, session**, and out objects are derived by accessing attributes of this object.

The pageContext object also contains information about the directives issued to the JSP page, including the buffering information, the errorPageURL, and page scope.

The PageContext class defines several fields, including **PAGE\_SCOPE, REQUEST\_SCOPE, SESSION\_SCOPE,** and **APPLICATION\_SCOPE**, which identify the four scopes. It also supports more than 40 methods, about half of which are inherited from the **javax.servlet.jsp.JspContext class**.

One of the important methods is **removeAttribute**. This method accepts either one or two arguments. For example, **pageContext.removeAttribute ("attrName")** removes the attribute from all scopes, while the following code only removes it from the page scope −

pageContext.removeAttribute("attrName", PAGE\_SCOPE);

The use of pageContext can be checked in [JSP - File Uploading](https://www.tutorialspoint.com/jsp/jsp_file_uploading.htm) chapter.

# UNIT - JavaBeans

A JavaBean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.

Following are the unique characteristics that distinguish a JavaBean from other Java classes −

* It provides a default, no-argument constructor.
* It should be serializable and that which can implement the **Serializable** interface.
* It may have a number of properties which can be read or written.
* It may have a number of "**getter**" and "**setter**" methods for the properties.

## JavaBeans Properties

A JavaBean property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define.

A JavaBean property may be **read, write, read only**, or **write only**. JavaBean properties are accessed through two methods in the JavaBean's implementation class −

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1 | get**PropertyName**()  For example, if property name is *firstName*, your method name would be **getFirstName()** to read that property. This method is called accessor. |
| 2 | set**PropertyName**()  For example, if property name is *firstName*, your method name would be **setFirstName()** to write that property. This method is called mutator. |

A read-only attribute will have only a **getPropertyName()** method, and a write-only attribute will have only a **setPropertyName()** method.

## JavaBeans Example

Consider a student class with few properties −

package com.tutorialspoint;

public class StudentsBean implements java.io.Serializable {

private String firstName = null;

private String lastName = null;

private int age = 0;

public StudentsBean() {

}

public String getFirstName(){

return firstName;

}

public String getLastName(){

return lastName;

}

public int getAge(){

return age;

}

public void setFirstName(String firstName){

this.firstName = firstName;

}

public void setLastName(String lastName){

this.lastName = lastName;

}

public void setAge(Integer age){

this.age = age;

}

}

## Accessing JavaBeans

The **useBean** action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows −

<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>

Here values for the scope attribute can be a **page, request, session** or **application based** on your requirement. The value of the **id** attribute may be any value as a long as it is a unique name among other **useBean declarations** in the same JSP.

Following example shows how to use the useBean action −

<html>

<head>

<title>useBean Example</title>

</head>

<body>

<jsp:useBean id = "date" class = "java.util.Date" />

<p>The date/time is <%= date %>

</body>

</html>

You will receive the following result − −

The date/time is Thu Sep 30 11:18:11 GST 2010

## Accessing JavaBeans Properties

Along with **<jsp:useBean...>** action, you can use the **<jsp:getProperty/>** action to access the get methods and the **<jsp:setProperty/>** action to access the set methods. Here is the full syntax −

<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">

<jsp:setProperty name = "bean's id" property = "property name"

value = "value"/>

<jsp:getProperty name = "bean's id" property = "property name"/>

...........

</jsp:useBean>

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the **get** or the **set** methods that should be invoked.

Following example shows how to access the data using the above syntax −

<html>

<head>

<title>get and set properties Example</title>

</head>

<body>

<jsp:useBean id = "students" class = "com.tutorialspoint.StudentsBean">

<jsp:setProperty name = "students" property = "firstName" value = "Zara"/>

<jsp:setProperty name = "students" property = "lastName" value = "Ali"/>

<jsp:setProperty name = "students" property = "age" value = "10"/>

</jsp:useBean>

<p>Student First Name:

<jsp:getProperty name = "students" property = "firstName"/>

</p>

<p>Student Last Name:

<jsp:getProperty name = "students" property = "lastName"/>

</p>

<p>Student Age:

<jsp:getProperty name = "students" property = "age"/>

</p>

</body>

</html>

Let us make the **StudentsBean.class** available in CLASSPATH. Access the above JSP. the following result will be displayed −

Student First Name: Zara

Student Last Name: Ali

Student Age: 10

**JAR File**

## What is JAR?

JAR stands for Java ARchive. It's a file format based on the popular ZIP file format and is used for aggregating many files into one. Although JAR can be used as a general archiving tool, the primary motivation for its development was so that Java applets and their requisite components (.class files, images and sounds) can be downloaded to a browser in a single HTTP transaction, rather than opening a new connection for each piece. This greatly improves the speed with which an applet can be loaded onto a web page and begin functioning. The JAR format also supports compression, which reduces the size of the file and improves download time still further. Additionally, individual entries in a JAR file may be digitally signed by the applet author to authenticate their origin.

JAR is:

* the only archive format that is cross-platform
* the only format that handles audio and image files as well as class files
* backward-compatible with existing applet code
* an open standard, fully extendable, and written in java
* the preferred way to bundle the pieces of a java applet

JAR consists of a zip archive, as defined by PKWARE, containing a manifest file and potentially signature files, as defined in the [JAR File Specification](https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html).

## The APPLET tag

Changing the APPLET tag in your HTML page to accomodate a JAR file is simple. The JAR file on the server is identified by the **ARCHIVE** parameter, where the directory location of the jar file should be relative to the location of the html page:

<applet code=Animator.class

archive="jars/animator.jar"

width=460 height=160>

<param name=foo value="bar">

</applet>

Note that the familiar **CODE=myApplet.class** parameter must still be present. The **CODE** parameter, as always, identifies the name of the applet where execution begins. However, the class file for the applet and all of its helper classes are loaded from the JAR file.

Th ARCHIVE attribute describes one or more JAR files containing classes and other resources that will be "preloaded". The classes are loaded using an instance of an AppletClassLoader with the given CODEBASE. It takes the form archive = archiveList. The archives in archiveList are separated by ",".

Once the archive file is identified, it is downloaded and separated into its components. During the execution of the applet, when a new class, image or audio clip is requested by the applet, it is searched for first in the archives associated with the applet. If the file is not found amongst the archives that were downloaded, it is searched for on the applet's server, relative to the CODEBASE (that is, it is searched for as in JDK1.0.2).

The archive tag may specify multiple JAR files. Each JAR file must be separated by "," (comma). Each file is downloaded in turn:

<applet code=Animator.class

archive="classes.jar , images.jar , sounds.jar"

width=460 height=160>

<param name=foo value="bar">

</applet>

There can be any amount of white space between entries within the archive parameter. In addition, the archive tag itself is case-insensitive; it can be lower-case, upper-case, or any combination of lower- and upper-case letters, such as ArCHiVe.

## Executable JAR Files

On Microsoft Windows systems, the Java 2 Runtime Environment's installation program will register a default association for JAR files so that double-clicking a JAR file on the desktop will automatically run it with javaw -jar. Dependent extensions bundled with the application will also be loaded automatically. This feature makes the end-user runtime environment easier to use on Microsoft Windows systems.

The Solaris 2.6 kernel has already been extended to recognize the special "magic" number that identifies a JAR file, and to invoke java -jar on such a JAR file as if it were a native Solaris executable. A application packaged in a JAR file can thus be executed directly from the command line or by clicking an icon on the CDE desktop.