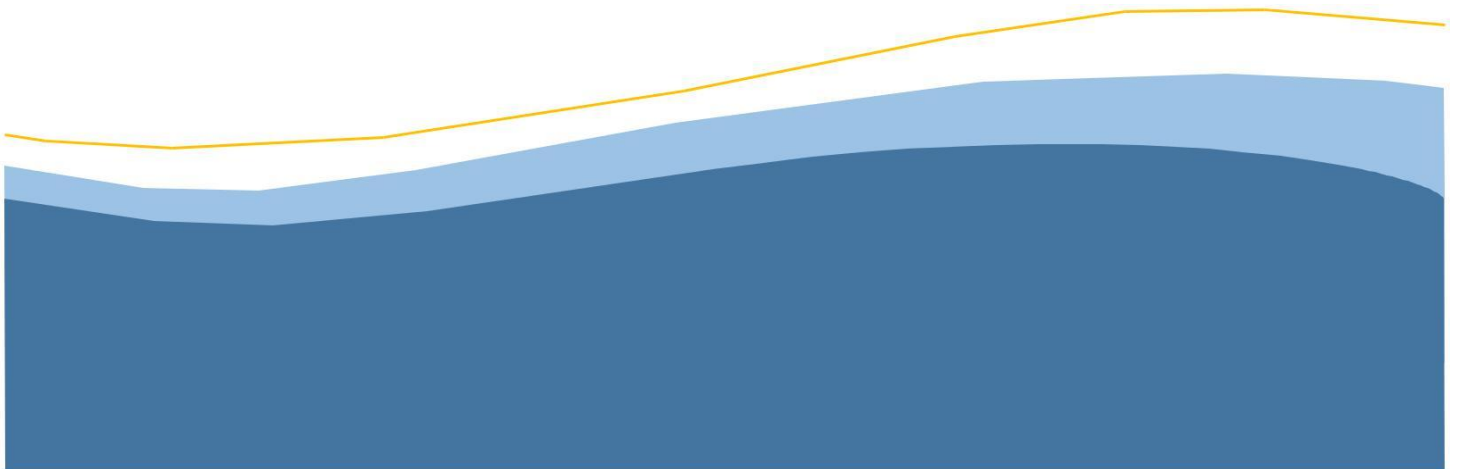


Project Based Learning in Java

CST-358/ITT-358





Program Educational Objectives (PEOs)


The graduates will

PEO1	Graduate will be able to serve professionally while engaging with a Government firm, industry, corporate, academic and research organization or by contributing being an entrepreneur.
PEO2	Graduate will be able to work effectively in different fields with a core expertise in analysis, design, networking, security, and development using advanced tools.
PEO3	Graduate will be able to develop themselves professionally by lifelong learning through innovation and research while benefiting the society.
PEO4	Graduate will be able to show the leadership in diverse cultures, nationalities and fields while working with interdisciplinary teams.
PEO5	Graduate will be committed team member through complacency and ethical values.

Program Outcomes (POs)

At the end of the B.E. (CSE) – graduates will be able to

PO1	Engineering Knowledge: Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
PO2	Problem analysis: Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.
PO3	Design/Development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety and the cultural, societal, and environmental considerations.
PO4	Investigations of complex problems: Use research based knowledge and research methods including design of experiments, analysis and interpretation of data and synthesis of the information to provide valid conclusions.
PO5	Modern tool usage: Create, select, and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.



Contents

Syllabus	4
Java Fundamentals	7
OOPS using JAVA	45
Exception Handling.....	57
Collection Framework	60
Multithreading	76
Wrapper Classes	92
JDBC	99
Servlets and JSP	148
XML and Web Services.....	177
Important contents beyond syllabus	179
References	194

Syllabus

CHANDIGARH UNIVERSITY, GHARUAN

Subject Code	Project Based Learning in JAVA	L	T	P	C
(CST-358/ITT-358)	Total Contact Hours : 30Hours	0	2	0	2
	Common to all Specializations of CSE 3 rd Year				
	Prerequisite: Knowing Programming Language Java				
Marks-100					
Internal-40			External-60		
Course Objectives					
<ul style="list-style-type: none">• Programming in the Java programming language.• Knowledge of object-oriented paradigm in the Java programming language.• The use of Java in a variety of technologies and on different platforms.• Understand the server side programming.					
Unit	Course Outcomes				
I	<ul style="list-style-type: none">• To gain knowledge of the structure and use the Java programming language for various technologies				
II	<ul style="list-style-type: none">• Annotations and Databases usage in project development				
III	<ul style="list-style-type: none">• Web Based Java Application Development				

Unit-I

Java Fundamentals: Introduction to Java. Difference between C++ and Java. Keywords, Tokens, Data types. Use of public, private and protected.

OOPS using Java: Use of class and method in Java. Inheritance, Abstraction, Polymorphism, Encapsulation and data privacy. Difference between method overloading and method overriding.

Exception Handling: Introduction to Exceptions. Difference between error and exception. Use of try, catch and throw. Difference between throw and throws. Types of Exceptions, Exception handling in Java.

Unit-II

Collection Framework: Use of Collections in Java. ArrayList, LinkedList, HashMap, TreeMap, HashSet in Java.. Multithreading in Java. Thread Synchronization. Thread Priority, Thread LifeCycle.

Wrapper Classes, I/O Streams and Annotations: Use of wrapper classes in Java- Integer, Character, Long, Boolean. Autoboxing and Unboxing. Byte stream, Character stream, Object serialization, cloning. System defined annotations, Custom annotations, application of annotations, Testing using JUnit.

JDBC: Database connectivity, Types of Drivers for connection, Connection Example. CRUD operations using Database, Configuring various types of drivers for Java Database Connectivity, MVC Model for project development, Sequence, Dual table , Date type management in Java.

Unit-III

Servlets and JSP: Servlet Lifecycle, Generic Servlet, Http Servlet, Linking Servlet to HTML, HttpServlet Request and Response, Servlet with JDBC, Configuring project using servlet, Servlet Config and Servlet Mapping JSP declaration, JSP directives, JSP Scriptlets, JSP include tag, JSP page tag, JSTL.

XML and Web Services: Structure of XML, Elements of XML 1.0, 2.0, DTDs, XML parser, DOM parser, Web services using REST and HTTP, Creating webservices for database access via remote servers

Text books:

1. Herbert Schildt, Java : The Complete Reference, 9th Edition, Oracle Press.
2. Gary Cornell, Core Java Volume II Advanced Features, 8th Edition, Pearson Education.
3. Jim Keogh, J2ee : Complete Reference, 1st Edition, Tata McGraw Hill.

Reference books:

1. James Gosling, Ken Arnold and David Holmes, Java Programming Language, 5th Edition, Pearson Education.
2. Gary Cornell, Core Java Volume I, 3rd Edition, Pearson Education.

Java Fundamentals

1.1 Important Features of Java

1. **Object Oriented:** In java everything is an Object. Java can be easily extended since it is based on the Object model.
2. **Platform independent:** Unlike many other programming languages including C and C++ when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.
3. **Simple:** Java is designed to be easy to learn. If you understand the basic concept of OOP java would be easy to master.
4. **Secure:** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
5. **Architectural- neutral:** Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence Java runtime system.
6. **Portable:** being architectural neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler and Java is written in ANSI C with a clean portability boundary which is a POSIX subset.
7. **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
8. **Multi-threaded:** With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
9. **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
10. **High Performance:** With the use of Just-In-Time compilers Java enables high performance.
11. **Distributed:** Java is designed for the distributed environment of the internet.**Dynamic :** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment.

Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

1.2 History of Java

James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May 2007 Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Tools you will need:

For performing the examples discussed in this tutorial, you will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128 MB of RAM recommended).

You also will need the following softwares:

Linux 7.1 or Windows 95/98/2000/XP operating system.

Java JDK 5

Microsoft Notepad or any other text editor

Java Environment Setup:

Java SE is freely available from the link [Download Java](#). So you download a version based on your operating system.

1.3 Fundamentals of Java

- a) Object** - Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
- b) Class** - A class can be defined as a template/ blue print that describe the behaviors/states that object of its type support.
- c) Methods** - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- d) Instant Variables** - Each object has its unique set of instant variables. An object's state is created by the values assigned to these instant variables.

First Java Program:

Let us look at a simple code that would print the words Hello World.

```
Public class MyFirstJavaProgram{  
/* This is my first java program.  
    * This will print 'Hello World' as the output  
    */  
publicstaticvoid main(String[]args){  
System.out.println("Hello World");// prints Hello World  
}  
}
```

About Java programs, it is very important to keep in mind the following points.

- e) Case Sensitivity** - Java is case sensitive which means identifier **Hello** and **hello** would have different meaning in Java.
- f) Class Names** - For all class names the first letter should be in Upper Case.

If several words are used to form a name of the class each inner words first letter should be in Upper Case.

Example class MyFirstJavaClass

g) Method Names - All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

Example public void myMethodName()

h) Program File Name - Name of the program file should exactly match the class name. When saving the file you should save it using the class name (Remember java is case sensitive) and append '.java' to the end of the name. (if the file name and the class name do not match your program will not compile).

Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as 'MyFirstJavaProgram.java'

public static void main(String args[]) - java program processing starts from the main() method which is a mandatory part of every java program..

i) Java Identifiers:

All java components require names. Names used for classes, variables and methods are called identifiers.

In java there are several points to remember about identifiers. They are as follows:

All identifiers should begin with a letter (A to Z or a to z), currency character (\$) or an underscore (_).

After the first character identifiers can have any combination of characters.

A key word cannot be used as an identifier.

Most importantly identifiers are case sensitive.

Examples of legal identifiers: age, \$salary, _value, __1_value

Examples of illegal identifiers: 123abc, -salary

Java Modifiers:

Like other languages it is possible to modify classes, methods etc by using modifiers. There are two categories of modifiers.

j) Access Modifiers: default, public, protected, private

k) Non-access Modifiers: final, abstract, strictfp

We will be looking into more details about modifiers in the next section.

l) Java Variables:

We would see following type of variables in Java:

Local Variables

Class Variables (Static Variables)

Instance Variables (Non static variables)

m) Java Arrays:

Arrays are objects that store multiple variables of the same type. However an Array itself is an object on the heap. We will look into how to declare, construct and initialize in the upcoming chapters.

n) Java Enums:

Enums were introduced in java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

With the use of enums it is possible to reduce the number of bugs in your code.

For example if we consider an application for a fresh juice shop it would be possible to restrict the glass size to small, medium and Large. This would make sure that it would not allow anyone to order any size other than the small, medium or large.

Example:

```
class FreshJuice{
```

```
enum FreshJuiceSize{ SIZE, MEDIUM, LARGE }
FreshJuiceSize size;
}
public class FreshJuiceTest{
public static void main(String args[]){
FreshJuice juice =new FreshJuice();
    juice.size =FreshJuice.FreshJuiceSize.MEDIUM ;
System.out.println("Size :"+ juice.size);
}
}
```

Note: enums can be declared as their own or inside a class. Methods, variables, constructors can be defined inside enums as well.

Java Keywords:

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
Do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface

long	native	new	package
private	protected	public	return
short	Static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	While		

O) Comments in Java

Java supports single line and multi-line comments very similar to c and c++. All characters available inside any comment are ignored by Java compiler.

```
public class MyFirstJavaProgram{

    /* This is my first java program.
       * This will print 'Hello World' as the output
       * This is an example of multi-line comments.
       */

    public static void main(String[] args){
        // This is an example of single line comment
        /* This is also an example of single line comment. */
        System.out.println("Hello World");
    }
}
```

p) Data Types in Java

There are two data types available in Java:

Primitive Data Types

Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a key word. Let us now look into detail about the eight primitive data types.

byte

short

int

long

float

double

boolean

char

Reference Data Types:

Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.

Class objects, and various type of array variables come under reference data type.

Default value of any reference variable is null.

A reference variable can be used to refer to any object of the declared type or any compatible type.

Example : `Animal animal = new Animal("giraffe");`

q) Java Literals:

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

```
byte a = 68;  
char a = 'A'
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"  
"two\nlines"  
 "\"This is in quotes\""
```

Java language supports few special escape sequences for String and char literals as well. They are:

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	Tab

\"	Double quote
\'	Single quote
\\	Backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Java Access Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

Visible to the package. the default. No modifiers are needed.

Visible to the class only (private).

Visible to the world (public).

Visible to the package and all subclasses (protected).

Java Basic Operators:

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

The Arithmetic Operators:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30

-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increase the value of operand by 1	B++ gives 21
--	Decrement - Decrease the value of operand by 1	B-- gives 19

The Relational Operators:

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.

>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000

>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

The Assignment Operators:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assigne value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result	C += A is equivalent

	to left operand	to $C = C + A$
<code>-=</code>	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	$C -= A$ is equivalent to $C = C - A$
<code>*=</code>	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	$C *= A$ is equivalent to $C = C * A$
<code>/=</code>	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	$C /= A$ is equivalent to $C = C / A$
<code>%=</code>	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	$C \% = A$ is equivalent to $C = C \% A$
<code><<=</code>	Left shift AND assignment operator	$C <<= 2$ is same as $C = C << 2$
<code>>>=</code>	Right shift AND assignment operator	$C >>= 2$ is same as $C = C >> 2$
<code>&=</code>	Bitwise AND assignment operator	$C \&= 2$ is same as $C = C \& 2$
<code>^=</code>	bitwise exclusive OR and assignment operator	$C \wedge= 2$ is same as $C = C \wedge 2$
<code> =</code>	bitwise inclusive OR and assignment operator	$C = 2$ is same as $C =$

r) Misc Operators

There are few other operators supported by Java Language.

Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as :

```
variable x =(expression)?valueiftrue: value iffalse
```

instanceOf Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is written as:

```
(Object reference variable ) instanceof (class/interface type)
```

Precedence of Java Operators:

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ -- ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>>>><<	Left to right

Relational	>>= <<=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

The syntax of a while loop is:

```
while(Boolean_expression)
{
//Statements
}
```

The do...while Loop:

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop is:

```
do
{
//Statements
}while(Boolean_expression);
```

The for Loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

The syntax of a for loop is:

```
for(initialization;Boolean_expression; update)
{
//Statements
}
```

Enhanced for loop in Java:

As of java 5 the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

The syntax of enhanced for loop is:

```
for(declaration : expression)
{
//Statements
}
```

The break Keyword:

The break keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

The continue Keyword:

The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

- In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

The syntax of a continue is a single statement inside any loop:

```
continue;
```

The if Statement:

An if statement consists of a Boolean expression followed by one or more statements.

Syntax:

The syntax of an if statement is:

```
if(Boolean_expression)
{
```



```
//Statements will execute if the Boolean expression is true
}
```

The if...else Statement:

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

Syntax:

The syntax of a if...else is:

```
if(Boolean_expression){
//Executes when the Boolean expression is true
}else{
//Executes when the Boolean expression is false
}
```

The if...else if...else Statement:

An if statement can be followed by an optional else if...else statement, which is very usefull to test various conditions using single if...else if statement.

Syntax:

The syntax of a if...else is:

```
if(Boolean_expression1){
//Executes when the Boolean expression 1 is true
}elseif(Boolean_expression2){
//Executes when the Boolean expression 2 is true
}elseif(Boolean_expression3){
//Executes when the Boolean expression 3 is true
}else{
//Executes when the one of the above condition is true.
}
```

Nested if...else Statement:

It is always legal to nest if-else statements. When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax for a nested if...else is as follows:

```
if(Boolean_expression1){  
    //Executes when the Boolean expression 1 is true  
    if(Boolean_expression2){  
        //Executes when the Boolean expression 2 is true  
    }  
}
```

The switch Statement:

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

The syntax of enhanced for loop is:

```
switch(expression){  
    case value :  
        //Statements  
        break;//optional  
    case value :  
        //Statements
```

```
break;//optional
//You can have any number of case statements.
default://Optional
//Statements
}
```

Java Methods:

A Java method is a collection of statements that are grouped together to perform an operation. When you call the `System.out.println` method, for example, the system actually executes several statements in order to display a message on the console.

In general, a method has the following syntax:

```
modifier returnType methodName(list of parameters){
// Method body;
}
```

A method definition consists of a method header and a method body. Here are all the parts of a method:

Modifiers: The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.

Return Type: A method may return a value. The `returnValueType` is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the `returnValueType` is the keyword **void**.

Method Name: This is the actual name of the method. The method name and the parameter list together constitute the method signature.

Parameters: A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.

Method Body: The method body contains a collection of statements that define what the method does.

Java Classes & Objects:

Object - Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.

Class - A class can be defined as a template/ blue print that describe the behaviors/states that object of its type support.

A sample of a class is given below:

```
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 instantiated when the class is loaded. 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.

Exceptions Handling:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
//Protected code
}catch(ExceptionName e1)
{
//Catch block
}
```

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
//Catch block
}
```

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword.

The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the difference in throws and throw keywords.

The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
//Catch block
}finally
{
//The finally block always executes.
}
```

1.4 File Handling in java

I/O package

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters, etc.

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to Files and networks but this tutorial covers very basic functionality related to streams and I/O. We would see most commonly used example one by one:

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file:

```
import java.io.*;

public class CopyFile {
    public static void main(String args[]) throws IOException
    {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("input.txt");
```

```
out=newFileOutputStream("output.txt");

int c;
while((c =in.read())!=-1){
out.write(c);
}
}finally{
if(in!=null){
in.close();
}
if(out!=null){
out.close();
}
}
}
```

Now let's have a file **input.txt** with the following content:

This is test for copy file.

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
```

```
$java CopyFile
```


Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, where as Java**Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are **FileReader** and **FileWriter**.. Though internally **FileReader** uses **FileInputStream** and **FileWriter** uses **FileOutputStream** but here major difference is that **FileReader** reads two bytes at a time and **FileWriter** writes two bytes at a time.

We can re-write above example which makes use of these two classes to copy an input file (having unicode characters) into an output file:

```
import java.io.*;

public class CopyFile {

    public static void main(String args[]) throws IOException
    {
        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("input.txt");
            out = new FileWriter("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
```

```
if(in!=null){  
in.close();  
}  
if(out!=null){  
out.close();  
}  
}  
}  
}
```

Now let's have a file **input.txt** with the following content:

This is test for copy file.

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
```

```
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similar way Java provides following three standard streams

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.

- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as **System.out**.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as **System.err**.

Following is a simple program which creates **InputStreamReader** to read standard input stream until the user types a "q":

```
import java.io.*;

public class ReadConsole {
    public static void main(String args[]) throws IOException
    {
        InputStreamReader cin = null;

        try {
            cin = new InputStreamReader(System.in);
            System.out.println("Enter characters, 'q' to quit.");
            char c;
            do {
                c = (char) cin.read();
                System.out.print(c);
            } while (c != 'q');
        } finally {
            if (cin != null) {
                cin.close();
            }
        }
    }
}
```

```
}  
  
}  
  
}  
  
}
```

Let's keep above code in ReadConsole.java file and try to compile and execute it as below. This program continues reading and outputting same character until we press 'q':

```
$javac ReadConsole.java
```

```
$java ReadConsole
```

```
Enter characters,'q' to quit.
```

```
l
```

```
l
```

```
e
```

```
e
```

```
q
```

```
q
```

Reading and Writing Files:

As described earlier, A stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.

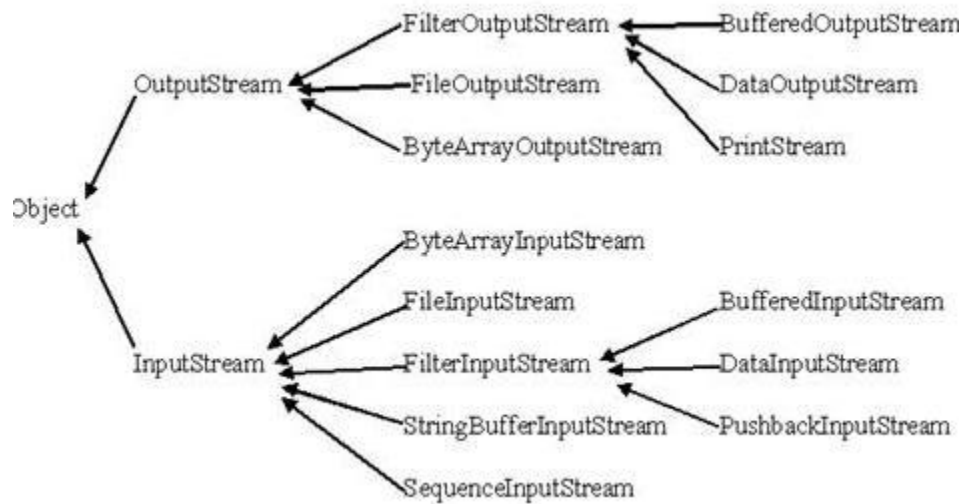


Fig 1.1 Hierarchy of classes of I/O streams

The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial:

FileInputStream:

This stream is used for reading data from the files. Objects can be created using the keyword `new` and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file.:

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using `File()` method as follows:

```
File f = new File("C:/java/hello");
```

```
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

SN	Methods with Description
----	--------------------------

1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public int read(int r)throws IOException{} This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
4	public int read(byte[] r) throws IOException{} This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
5	public int available() throws IOException{} Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links:

- [ByteArrayInputStream](#)
- [DataInputStream](#)

FileOutputStream:

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a `FileOutputStream` object.

Following constructor takes a file name as a string to create an input stream object to write the file:

```
OutputStream f =newFileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using `File()` method as follows:

```
File f =newFile("C:/java/hello");  
OutputStream f =newFileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

SN	Methods with Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an <code>IOException</code>
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an <code>IOException</code> .
3	public void write(int w)throws IOException{} This methods writes the specified byte to the output stream.
4	public void write(byte[] w)

Writes w.length bytes from the mentioned byte array to the OutputStream.
--

There are other important output streams available, for more detail you can refer to the following links:

- [ByteArrayOutputStream](#)
- [DataOutputStream](#)

Example:

Following is the example to demonstrate InputStream and OutputStream:

```
import java.io.*;

public class FileStreamTest{

    public static void main(String args[]){

        try{
            byte bWrite []={ 11,21,3,40,5};
            OutputStream os =new FileOutputStream("test.txt");
            for(int x=0; x < bWrite.length ; x++){
                os.write( bWrite[x]);// writes the bytes
            }
            os.close();

            InputStream is=new FileInputStream("test.txt");
            int size =is.available();
```



```
for(int i=0; i< size; i++){  
    System.out.print((char)is.read()+" ");  
}  
is.close();  
}catch(IOException e){  
    System.out.print("Exception");  
}  
}  
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be output on the stdout screen.

File Navigation and I/O:

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java:

A directory is a File which can contains a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories:

There are two useful **File** utility methods, which can be used to create directories:

- The **mkdir()** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory:

```
import java.io.File;

public class CreateDir {
    public static void main(String args[]) {
        String dirname = "/tmp/user/java/bin";
        File d = new File(dirname);
        // Create directory now.
        d.mkdirs();
    }
}
```

Compile and execute above code to create "/tmp/user/java/bin".

Note: Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories:

You can use **list()** method provided by **File** object to list down all the files and directories available in a directory as follows:

```
import java.io.File;
```

```

public class ReadDir{

    public static void main(String[] args){

        File file =null;

        String[] paths;

        try{

            // create new file object

            file=newFile("/tmp");

            // array of files and directory

            paths= file.list();

            // for each name in the path array

            for(String path:paths)

            {

                // prints filename and directory name

                System.out.println(path);

            }

        }catch(Exception e){

            // if any error occurs

            e.printStackTrace();

        }

    }

}

```

```
}  
  
}
```

This would produce following result based on the directories and files available in your **/tmp** directory:

test1.txt

test2.txt

ReadDir.java

ReadDir.class

OOPS using JAVA

1.5 Classes and Objects in Java

Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

Class

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers** : A class can be public or has default access (Refer this for details).
2. **Class name**: The name should begin with a initial letter (capitalized by convention).
3. **Superclass(if any)**: The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces(if any)**: A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body**: The class body surrounded by braces, { }.

Constructors are used for initializing new objects. Fields are variables that provides the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

There are various types of classes that are used in real time applications such as nested classes, anonymous classes, lambda expressions.

Object

It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State** : It is represented by attributes of an object. It also reflects the properties of an object.

2. Behavior : It is represented by methods of an object. It also reflects the response of an object with other objects.
3. Identity : It gives a unique name to an object and enables one object to interact with other objects.

Example of an object : dog

Objects correspond to things found in the real world. For example, a graphics program may have objects such as “circle”, “square”, “menu”. An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Example :

As we declare variables like (type name;). This notifies the compiler that we will use name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. So for reference variable, type must be strictly a concrete class name. In general, we can't create objects of an abstract class or an interface.

Dog tuffy;

If we declare reference variable(tuffy) like this, its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

Initializing an object

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

// Class Declaration

```

public class Dog
{
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public Dog(String name, String breed,
                int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // method 1
    public String getName()
    {
        return name;
    }

    // method 2
    public String getBreed()
    {
        return breed;
    }
}

```

```

// method 3
public int getAge()
{
    return age;
}

// method 4
public String getColor()
{
    return color;
}

@Override
public String toString()
{
    return("Hi my name is " + this.getName()+
        ".\nMy breed,age and color are " +
        this.getBreed()+", " + this.getAge()+
        ", "+ this.getColor());
}

public static void main(String[] args)
{
    Dog tuffy = new Dog("tuffy","papillon", 5, "white");
    System.out.println(tuffy.toString());
}
}

```

Output:

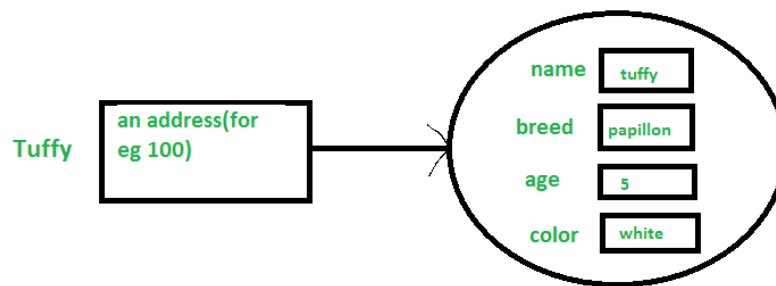
Hi my name is tuffy.

My breed,age and color are papillon,5,white

This class contains a single constructor. We can recognize a constructor because its declaration uses the same name as the class and it has no return type. The Java compiler differentiates the constructors based on the number and the type of the arguments. The constructor in the *Dog* class takes four arguments. The following statement provides “tuffy”, “papillon”, 5, “white” as values for those arguments:

```
Dog tuffy = new Dog("tuffy", "papillon", 5, "white");
```

The result of executing this statement can be illustrated as :



Note : All classes have at least **one** constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, also called the default constructor. This default constructor calls the class parent’s no-argument constructor (as it contain only one statement i.e `super();`), or the *Object* class constructor if the class has no other parent (as *Object* class is parent of all classes either directly or indirectly).

Ways to create object of a class

There are four ways to create objects in java. Strictly speaking there is only one way (by using *new* keyword), and the rest internally use *new* keyword.

Using new keyword : It is the most common and general way to create object in java. Example:

```
// creating object of class Test
```

```
Test t = new Test();
```

Using Class.forName(String className) method : There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the Class object associated with the class with the given string name. We have to give the fully qualified name for a class. On calling new Instance() method on this Class object returns new instance of the class with the given string name.

```
// creating object of public class Test
```

```
// consider class Test present in com.pl package
```

```
Test obj = (Test)Class.forName("com.pl.Test").newInstance();
```

Using clone() method: clone() method is present in Object class. It creates and returns a copy of the object.

```
// creating object of class Test
```

```
Test t1 = new Test();
```

```
// creating clone of above object
```

```
Test t2 = (Test)t1.clone();
```

Deserialization : De-serialization is technique of reading an object from the saved state in a file. Refer Serialization/De-Serialization in java

```
FileInputStream file = new FileInputStream(filename);
```

```
ObjectInputStream in = new ObjectInputStream(file);
```

```
Object obj = in.readObject();
```

Creating multiple objects by one type only (A good practice)

In real-time, we need different objects of a class in different methods. Creating a number of references for storing them is not a good practice and therefore we declare a static reference variable and use it whenever required. In this case, wastage of memory is less. The objects that are not referenced anymore will be destroyed by Garbage Collector of java. Example:

```
Test test = new Test();
```

```
test = new Test();
```

In inheritance system, we use parent class reference variable to store a sub-class object. In this case, we can switch into different subclass objects using same referenced variable. Example:

```
class Animal { }
```

```
class Dog extends Animal { }
```

```
class Cat extends Animal { }
```

```
public class Test
```

```
{
```

```
    // using Dog object
```

```
    Animal obj = new Dog();
```

```
    // using Cat object
```

```
    obj = new Cat();
```

```
}
```

Methods in Java

A method is a collection of statements that perform some specific task and return the result to the caller.

A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++, and Python.

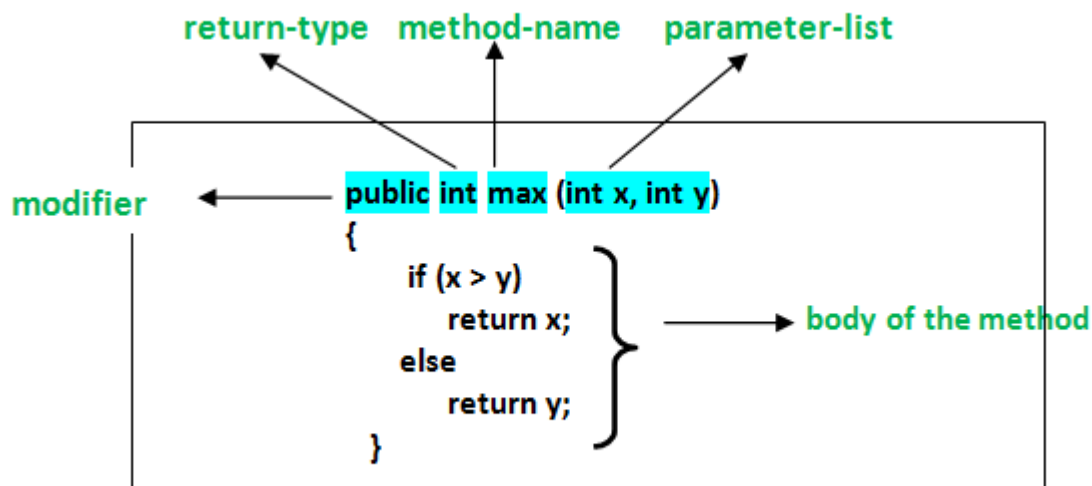
Methods are **time savers** and help us to **reuse** the code without retyping the code.

Method Declaration

In general, method declarations has six components :

- **Modifier-:** Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.
 - public: accessible in all class in your application.
 - protected: accessible within the class in which it is defined and in its **subclass(es)**

- **private**: accessible only within the class in which it is defined.
- **default** (declared/defined without using any modifier) : accessible within same class and package within which its class is defined.
- **The return type** : The data type of the value returned by the method or void if does not return a value.
- **Method Name** : the rules for field names apply to method names as well, but the convention is a little different.
- **Parameter list** : Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
- **Exception list** : The exceptions you expect by the method can throw, you can specify these exception(s).
- **Method body** : it is enclosed between braces. The code you need to be executed to perform your intended operations.



Method signature: It consists of the method name and a parameter list (number of parameters, type of the parameters and order of the parameters). The return type and exceptions are not considered as part of it.

Method Signature of above function:

`max(int x, int y)`

How to name a Method?: A method name is typically a single word that should be a **verb** in lowercase or multi-word, that begins with a **verb** in lowercase followed by **adjective, noun.....** After the first word, first letter of each word should be capitalized. For example, findSum, computeMax, setX and getX

Generally, A method has a unique name within the class in which it is defined but sometime a method might have the same name as other method names within the same class as method overloading is allowed in Java.

Calling a method

The method needs to be called for using its functionality. There can be three situations when a method is called:

A method returns to the code that invoked it when:

- It completes all the statements in the method
- It reaches a return statement
- Throws an exception

// Program to illustrate methods in java

```
import java.io.*;
```

```
class Addition {
```

```
    int sum = 0;
```

```
    public int addTwoInt(int a, int b){
```

```
        // adding two integer value.
```

```
        sum = a + b;
```

```
        //returning summation of two values.
```

```
        return sum;
```

```
    }
```

```
}
```

```

class GFG {
    public static void main (String[] args) {
        // creating an instance of Addition class
        Addition add = new Addition();
        // calling addTwoInt() method to add two integer using instance created
        // in above step.
        int s = add.addTwoInt(1,2);
        System.out.println("Sum of two integer values :"+ s);
    }
}

```

Output :

Sum of two integer values :3

See the below example to understand method call in detail :

```

// Java program to illustrate different ways of calling a method
import java.io.*;

```

```

class Test
{
    public static int i = 0;
    // constructor of class which counts
    //the number of the objects of the class.
    Test()
    {
        i++;
    }
    // static method is used to access static members of the class
    // and for getting total no of objects
    // of the same class created so far
    public static int get()

```

```

{
    // statements to be executed....
    return i;
}

// Instance method calling object directly
// that is created inside another class 'GFG'.
// Can also be called by object directly created in the same class
// and from another method defined in the same class
// and return integer value as return type is int.
public int m1()
{
    System.out.println("Inside the method m1 by object of GFG class");

    // calling m2() method within the same class.
    this.m2();

    // statements to be executed if any
    return 1;
}

// It doesn't return anything as
// return type is 'void'.
public void m2()
{
    System.out.println("In method m2 came from method m1");
}
}

class GFG

```

```

{
    public static void main(String[] args)
    {
        // Creating an instance of the class
        Test obj = new Test();

        // Calling the m1() method by the object created in above step.
        int i = obj.m1();
        System.out.println("Control returned after method m1 :" + i);

        // Call m2() method
        // obj.m2();
        int no_of_objects = Test.get();

        System.out.print("No of instances created till now : ");
        System.out.println(no_of_objects);

    }
}

```

Output :

Inside the method m1 by object of GFG class

In method m2 came from method m1

Control returned after method m1 :1

No of instances created till now : 1

Exception Handling

1.6 Exceptions Handling

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
//Protected code
}catch(ExceptionName e1)
{
//Catch block
}
```

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
```

```
//Catch block  
}
```

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the difference in throws and throw keywords.

The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try  
{  
//Protected code  
}catch(ExceptionType1 e1)  
{  
//Catch block  
}catch(ExceptionType2 e2)  
{  
//Catch block  
}catch(ExceptionType3 e3)  
{  
//Catch block  
}finally
```

```
{  
//The finally block always executes.  
}
```

Collection Framework

1.7 Collections in Java

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

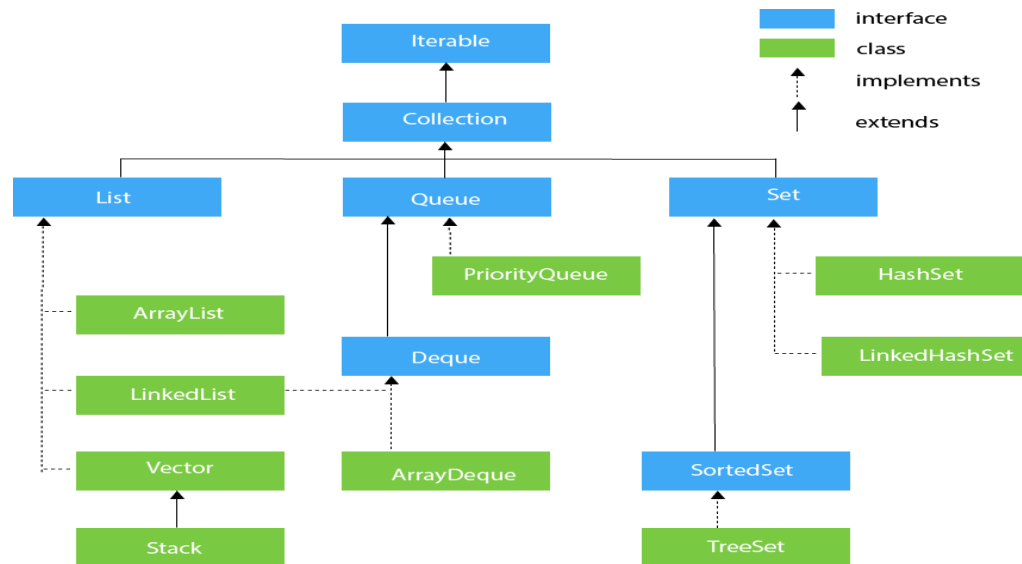
What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.



Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.

3	<code>public void remove()</code>	It removes the last elements returned by the iterator. It is less used.
---	-----------------------------------	---

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. `Iterator<T> iterator()`

It returns the iterator over the elements of type T.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are `Boolean add (Object obj)`, `Boolean addAll (Collection c)`, `void clear()`, etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes `ArrayList`, `LinkedList`, `Vector`, and `Stack`.

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;

class TestJavaCollection1 {

    public static void main(String args[]){

        ArrayList<String> list=new ArrayList<String>();//Creating arraylist

        list.add("Ravi");//Adding object in arraylist

        list.add("Vijay");

        list.add("Ravi");

        list.add("Ajay");

        //Traversing list through Iterator

        Iterator itr=list.iterator();
```

```

while(itr.hasNext()){

    System.out.println(itr.next());

}

}

}

```

Output:

Ravi

Vijay

Ravi

Ajay

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```

import java.util.*;

public class TestJavaCollection2{

    public static void main(String args[]){

        LinkedList<String> al=new LinkedList<String>();

        al.add("Ravi");

        al.add("Vijay");

        al.add("Ravi");
    }
}

```



```

al.add("Ajay");

Iterator<String> itr=al.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}

```

Output:

Ravi

Vijay

Ravi

Ajay

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```

import java.util.*;

public class TestJavaCollection3{

public static void main(String args[]){

Vector<String> v=new Vector<String>();

v.add("Ayush");

v.add("Amit");

```

```

v.add("Ashish");

v.add("Garima");

Iterator<String> itr=v.iterator();

while(itr.hasNext()){

    System.out.println(itr.next());

}

}

}

```

Output:

Ayush

Amit

Ashish

Garima

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```

import java.util.*;

public class TestJavaCollection4{

    public static void main(String args[]){

```

```
Stack<String> stack = new Stack<String>();

stack.push("Ayush");

stack.push("Garvit");

stack.push("Amit");

stack.push("Ashish");

stack.push("Garima");

stack.pop();

Iterator<String> itr=stack.iterator();

while(itr.hasNext()){

    System.out.println(itr.next());

}

}

}
```

Output:

Ayush

Garvit

Amit

Ashish

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

1. `Queue<String> q1 = new PriorityQueue();`
2. `Queue<String> q2 = new ArrayDeque();`

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection5{

    public static void main(String args[]){

        PriorityQueue<String> queue=new PriorityQueue<String>();

        queue.add("Amit Sharma");

        queue.add("Vijay Raj");

        queue.add("JaiShankar");

        queue.add("Raj");

        System.out.println("head:"+queue.element());

        System.out.println("head:"+queue.peek());
```

```
System.out.println("iterating the queue elements:");
```

```
Iterator itr=queue.iterator();
```

```
while(itr.hasNext()){
```

```
System.out.println(itr.next());
```

```
}
```

```
queue.remove();
```

```
queue.poll();
```

```
System.out.println("after removing two elements:");
```

```
Iterator<String> itr2=queue.iterator();
```

```
while(itr2.hasNext()){
```

```
System.out.println(itr2.next());
```

```
}
```

```
}
```

```
}
```

Output:

head:Amit Sharma

head:Amit Sharma

iterating the queue elements:

Amit Sharma

Raj

JaiShankar

Vijay Raj

after removing two elements:

Raj

Vijay Raj

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1. Deque d = **new** ArrayDeque();

1.8 ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection6{

    public static void main(String[] args) {

        //Creating Deque and adding elements

        Deque<String> deque = new ArrayDeque<String>();

        deque.add("Gautam");
```

```
deque.add("Karan");

deque.add("Ajay");

//Traversing elements

for (String str : deque) {

    System.out.println(str);

}

}

}
```

Output:

Gautam

Karan

Ajay

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection7{

public static void main(String args[]){

    //Creating HashSet and adding elements

    HashSet<String> set=new HashSet<String>();

    set.add("Ravi");

    set.add("Vijay");

    set.add("Ravi");

    set.add("Ajay");

    //Traversing elements

    Iterator<String> itr=set.iterator();

    while(itr.hasNext()){

        System.out.println(itr.next());

    }

}

}
```

Output:

Vijay

Ravi

Ajay

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection8{

public static void main(String args[]){

    LinkedHashSet<String> set=new LinkedHashSet<String>();

    set.add("Ravi");

    set.add("Vijay");

    set.add("Ravi");

    set.add("Ajay");

    Iterator<String> itr=set.iterator();

    while(itr.hasNext()){

        System.out.println(itr.next());

    }

}

}
```

Output:

Ravi

Vijay

Ajay

SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

1. SortedSet<data-type> set = **new** TreeSet();

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
import java.util.*;

public class TestJavaCollection9{

    public static void main(String args[]){

        //Creating and adding elements

        TreeSet<String> set=new TreeSet<String>();

        set.add("Ravi");
```

```
set.add("Vijay");  
  
set.add("Ravi");  
  
set.add("Ajay");  
  
//traversing elements  
  
Iterator<String> itr=set.iterator();  
  
while(itr.hasNext()){  
  
    System.out.println(itr.next());  
  
    }  
  
    }  
  
    }
```

Output:

Ajay

Ravi

Vijay

1.8 Multithreading

Java is a *multithreaded programming language* which means we can develop multithreaded program using Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.

By definition multitasking is when multiple processes share common processing resources such as a CPU. Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.

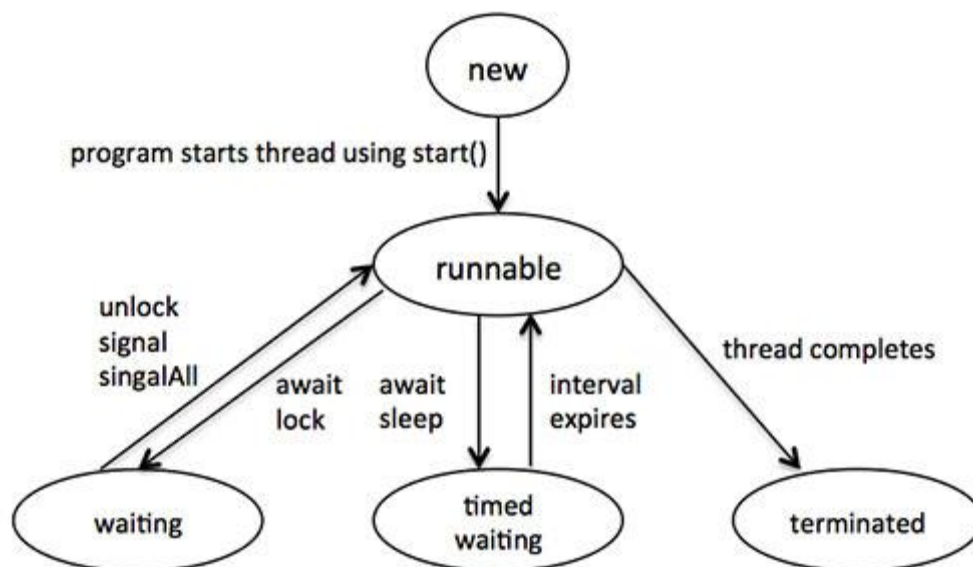


Fig 1.2: Life cycle of thread

Above-mentioned stages are explained here:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between `MIN_PRIORITY` (a constant of 1) and `MAX_PRIORITY` (a constant of 10). By default, every thread is given priority `NORM_PRIORITY` (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.

Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing **Runnable** interface. You will need to follow three basic steps:

Step 1:

As a first step you need to implement a `run()` method provided by **Runnable** interface. This method provides entry point for the thread and you will put your complete business logic inside this method. Following is simple syntax of `run()` method:

```
public void run()
```

Step 2:

At second step you will instantiate a **Thread** object using the following constructor:

```
Thread(Runnable threadObj, String threadName);
```

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3

Once Thread object is created, you can start it by calling **start()** method, which executes a call to `run()` method. Following is simple syntax of `start()` method:

```
void start();
```

Example:

Here is an example that creates a new thread and starts it running:

```
class RunnableDemo implements Runnable {  
    private Thread t;  
    private String threadName;  
  
    RunnableDemo(String name) {  
        threadName = name;  
    }  
}
```

```

System.out.println("Creating "+ threadName );

}

public void run(){

System.out.println("Running "+ threadName );

try{

for(int i =4; i >0; i--){

System.out.println("Thread: "+ threadName +", "+ i);

// Let the thread sleep for a while.

Thread.sleep(50);

}

}catch(InterruptedException e){

System.out.println("Thread "+ threadName +" interrupted.");

}

System.out.println("Thread "+ threadName +" exiting.");

}


public void start ()

{

System.out.println("Starting "+ threadName );

if(t ==null)

{

    t =new Thread(this, threadName);

    t.start ();

```

```
}  
  
}  
  
}  
  
public class TestThread {  
    public static void main(String args[]) {  
  
        RunnableDemo R1 = new RunnableDemo("Thread-1");  
        R1.start();  
  
        RunnableDemo R2 = new RunnableDemo("Thread-2");  
        R2.start();  
    }  
}
```

This would produce the following result:

```
CreatingThread-1  
StartingThread-1  
CreatingThread-2  
StartingThread-2  
RunningThread-1  
Thread:Thread-1,4
```



```
RunningThread-2
Thread:Thread-2,4
Thread:Thread-1,3
Thread:Thread-2,3
Thread:Thread-1,2
Thread:Thread-2,2
Thread:Thread-1,1
Thread:Thread-2,1
ThreadThread-1 exiting.
ThreadThread-2 exiting.
```

Create Thread by Extending Thread Class:

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override **run()** method available in Thread class. This method provides entry point for the thread and you will put your complete business logic inside this method. Following is simple syntax of run() method:

```
public void run()
```

Step 2

Once Thread object is created, you can start it by calling **start()** method, which executes a call to run() method. Following is simple syntax of start() method:

```
void start();
```

Example:

Here is the preceding program rewritten to extend Thread:

```
class ThreadDemo extends Thread {  
  
    private Thread t;  
  
    private String threadName;  
  
    ThreadDemo(String name) {  
  
        threadName = name;  
  
        System.out.println("Creating " + threadName);  
  
    }  
  
    public void run() {  
  
        System.out.println("Running " + threadName);  
  
        try {  
  
            for (int i = 4; i > 0; i--) {  
  
                System.out.println("Thread: " + threadName + ", " + i);  
  
                // Let the thread sleep for a while.  
  
                Thread.sleep(50);  
  
            }  
  
        } catch (InterruptedException e) {  
  
            System.out.println("Thread " + threadName + " interrupted.");  
  
        }  
  
        System.out.println("Thread " + threadName + " exiting.");  
    }  
}
```

```

}

public void start ()
{
    System.out.println("Starting "+ threadName );
    if(t==null)
    {
        t=new Thread(this, threadName);
        t.start ();
    }
}

}

}

}

public class TestThread{
    public static void main(String args[]){

        ThreadDemo T1 =new ThreadDemo("Thread-1");
        T1.start();

        ThreadDemo T2 =new ThreadDemo("Thread-2");
        T2.start();
    }
}

```

```
}
```

This would produce the following result:

```
CreatingThread-1
StartingThread-1
CreatingThread-2
StartingThread-2
RunningThread-1
Thread:Thread-1,4
RunningThread-2
Thread:Thread-2,4
Thread:Thread-1,3
Thread:Thread-2,3
Thread:Thread-1,2
Thread:Thread-2,2
Thread:Thread-1,1
Thread:Thread-2,1
ThreadThread-1 exiting.
ThreadThread-2 exiting.
```

Thread Methods:

Following is the list of important methods available in the Thread class.

SN	Methods with Description
1	<p>public void start()</p> <p>Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.</p>
2	<p>public void run()</p> <p>If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.</p>
3	<p>public final void setName(String name)</p> <p>Changes the name of the Thread object. There is also a getName() method for retrieving the name.</p>
4	<p>public final void setPriority(int priority)</p> <p>Sets the priority of this Thread object. The possible values are between 1 and 10.</p>
5	<p>public final void setDaemon(boolean on)</p> <p>A parameter of true denotes this Thread as a daemon thread.</p>
6	<p>public final void join(long millisec)</p> <p>The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.</p>

7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

SN	Methods with Description
1	public static void yield() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	public static void sleep(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds.
3	public static boolean holdsLock(Object x) Returns true if the current thread holds the lock on the given Object.
4	public static Thread currentThread()

	Returns a reference to the currently running thread, which is the thread that invokes this method.
5	public static void dumpStack() Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example:

The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable**:

```
// File Name : DisplayMessage.java

// Create a thread to implement Runnable

public class DisplayMessage implements Runnable
{
    private String message;

    public DisplayMessage(String message)
    {
        this.message = message;
    }

    public void run()
    {
        while(true)
        {
            System.out.println(message);
```

```
}  
  
}  
  
}
```

Following is another class which extends Thread class:

```
// File Name : GuessANumber.java  
  
// Create a thread to extend Thread  
  
public class GuessANumber extends Thread  
{  
    private int number;  
    public GuessANumber(int number)  
    {  
        this.number = number;  
    }  
    public void run()  
    {  
        int counter = 0;  
        int guess = 0;  
        do  
        {  
            guess = (int)(Math.random()*100+1);  
            System.out.println(this.getName()  
            +" guesses "+ guess);
```



```
counter++;

}while(guess != number);

System.out.println("** Correct! "+this.getName()
+" in "+ counter +" guesses.**");

}

}
```

Following is the main program which makes use of above defined classes:

```
// File Name : ThreadClassDemo.java

public class ThreadClassDemo
{
    public static void main(String[] args)
    {
        Runnable hello = new DisplayMessage("Hello");
        Thread thread1 = new Thread(hello);
        thread1.setDaemon(true);
        thread1.setName("hello");
        System.out.println("Starting hello thread...");
        thread1.start();

        Runnable bye = new DisplayMessage("Goodbye");
        Thread thread2 = new Thread(bye);
        thread2.setPriority(Thread.MIN_PRIORITY);
```

```

thread2.setDaemon(true);

System.out.println("Starting goodbye thread...");

thread2.start();


System.out.println("Starting thread3...");

Thread thread3 =newGuessANumber(27);

thread3.start();

try
{
thread3.join();

}catch(InterruptedException e)

{

System.out.println("Thread interrupted.");

}

System.out.println("Starting thread4...");

Thread thread4 =newGuessANumber(75);


        thread4.start();

System.out.println("main() is ending...");

}

}

```

This would produce the following result. You can try this example again and again and you would get different result every time.

Starting hello thread...

Starting goodbye thread...

Hello

Hello

Hello

Hello

Hello

Hello

Goodbye

Goodbye

Goodbye

Goodbye

Goodbye

Wrapper Classes

1.9 Wrapper classes in Java

The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the `valueOf()` method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1 {
    public static void main(String args[]){
        //Converting int into Integer
        int a=20;
        Integer i=Integer.valueOf(a);//converting int into Integer explicitly
        Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

        System.out.println(a+" "+i+" "+j);
    }
}
```

Output:

```
20 20 20
```

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the `intValue()` method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

```
3 3 3
```

Java Wrapper classes Example

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa
public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
```

```
float f=50.0F;  
double d=60.0D;  
char c='a';  
boolean b2=true;
```

```
//Autoboxing: Converting primitives into objects
```

```
Byte byteobj=b;  
Short shortobj=s;  
Integer intobj=i;  
Long longobj=l;  
Float floatobj=f;  
Double doubleobj=d;  
Character charobj=c;  
Boolean boolobj=b2;
```

```
//Printing objects
```

```
System.out.println("---Printing object values---");  
System.out.println("Byte object: "+byteobj);  
System.out.println("Short object: "+shortobj);  
System.out.println("Integer object: "+intobj);  
System.out.println("Long object: "+longobj);  
System.out.println("Float object: "+floatobj);  
System.out.println("Double object: "+doubleobj);  
System.out.println("Character object: "+charobj);  
System.out.println("Boolean object: "+boolobj);
```

```
//Unboxing: Converting Objects to Primitives
```

```
byte bytevalue=byteobj;  
short shortvalue=shortobj;  
int intvalue=intobj;  
long longvalue=longobj;
```

```
float floatvalue=floatobj;  
double doublevalue=doubleobj;  
char charvalue=charobj;  
boolean boolvalue=boolobj;
```

```
//Printing primitives  
System.out.println("---Printing primitive values---");  
System.out.println("byte value: "+bytevalue);  
System.out.println("short value: "+shortvalue);  
System.out.println("int value: "+intvalue);  
System.out.println("long value: "+longvalue);  
System.out.println("float value: "+floatvalue);  
System.out.println("double value: "+doublevalue);  
System.out.println("char value: "+charvalue);  
System.out.println("boolean value: "+boolvalue);  
}}
```

Output:

```
---Printing object values---  
Byte object: 10  
Short object: 20  
Integer object: 30  
Long object: 40  
Float object: 50.0  
Double object: 60.0  
Character object: a  
Boolean object: true  
---Printing primitive values---  
byte value: 10  
short value: 20  
int value: 30
```



```
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

Custom Wrapper class in Java

Java Wrapper classes wrap the primitive data types, that is why it is known as wrapper classes. We can also create a class which wraps a primitive data type. So, we can create a custom wrapper class in Java.

```
//Creating the custom wrapper class
class Javatpoint{
private int i;
Javatpoint(){ }
Javatpoint(int i){
this.i=i;
}
public int getValue(){
return i;
}
public void setValue(int i){
this.i=i;
}
@Override
public String toString() {
return Integer.toString(i);
}
}
//Testing the custom wrapper class
public class TestJavatpoint{
public static void main(String[] args){
```

```
Javatpoint j=new Javatpoint(10);  
System.out.println(j);  
}}
```

Output:

```
10
```

JDBC

1.10 What is JDBC?

JDBC stands for **J**ava **D**atabase **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.

1. 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 –

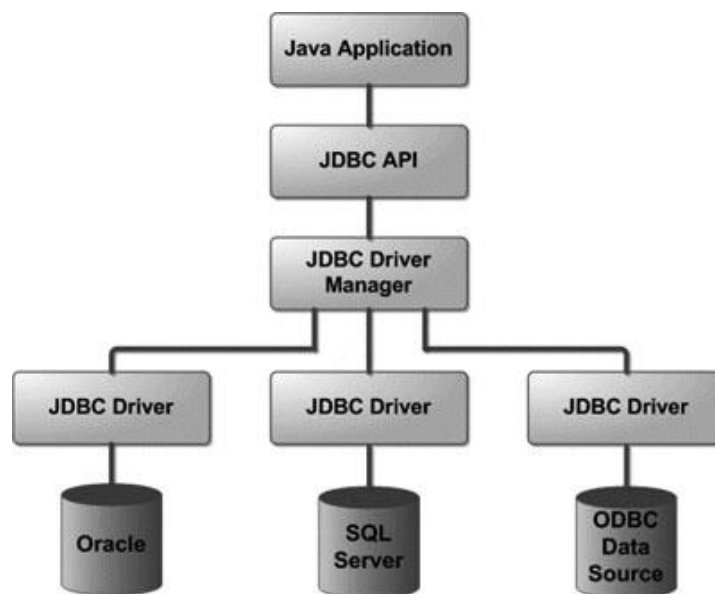


Fig: JDBC drivers

2. 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.

3. Structured Query Language (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, **Read**, **Update**, and **Delete** (often referred to as **CRUD** operations) data from a database.

For a detailed understanding on SQL, you can read our **MySQL Tutorial**.

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;
```

To start developing with JDBC, you should setup your JDBC environment

4. 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.

5. 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 past 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
```

```

staticfinalString JDBC_DRIVER ="com.mysql.jdbc.Driver";

staticfinalString DB_URL ="jdbc:mysql://localhost/EMP";


// Database credentials

staticfinalString USER ="username";

staticfinalString PASS ="password";


publicstaticvoid 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");

Stringlast=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:\>
```

6. What is JDBC Driver?

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server.

For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

The *Java.sql* package that ships with JDK, contains various classes with their behaviours defined and their actual implementations are done in third-party drivers. Third party vendors implements the *java.sql.Driver* interface in their database driver.

JDBC Drivers Types

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

Type 1: JDBC-ODBC Bridge Driver

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

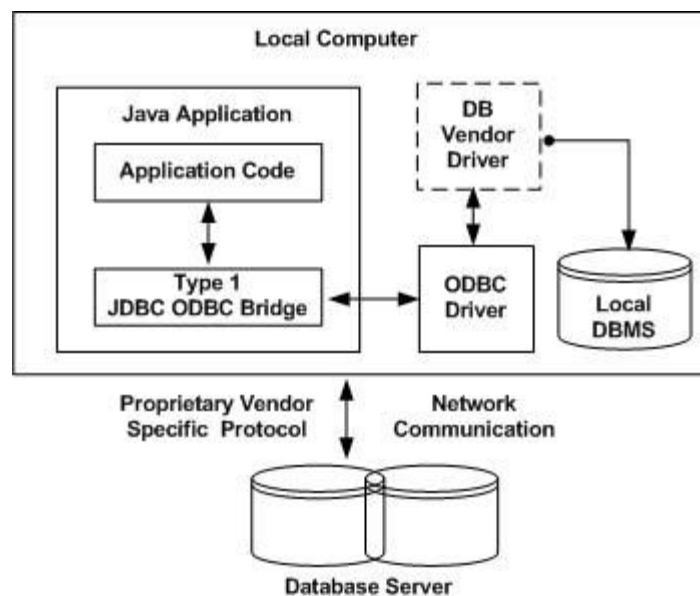


Fig: Type 1 driver

The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 2: JDBC-Native API

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

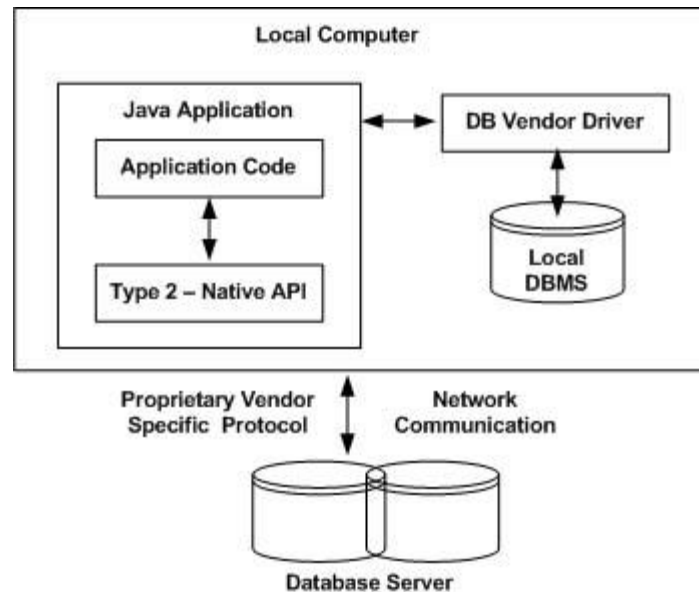


Fig: Type 2 driver

The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3: JDBC-Net pure Java

In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

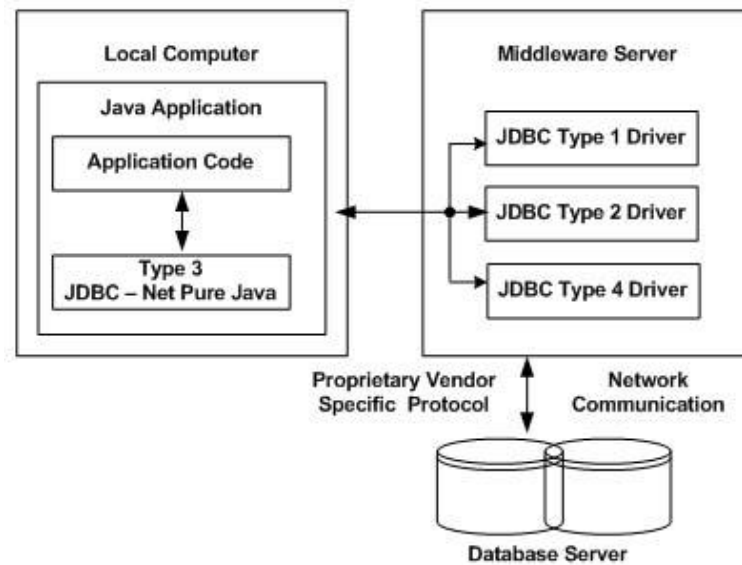


Fig: Type 3 drivers

You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

Type 4: 100% Pure Java

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

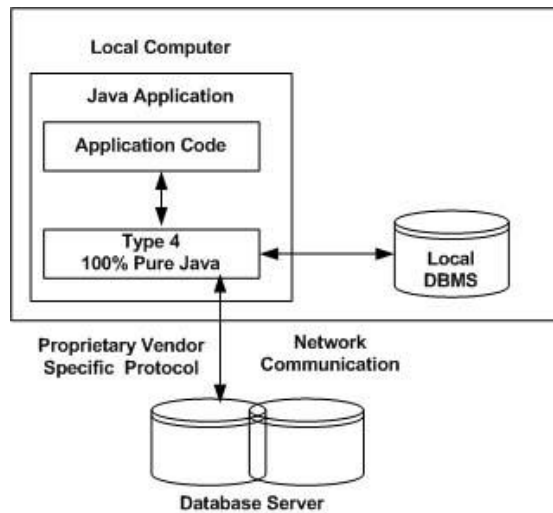


Fig: Type 4 drivers

MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Which Driver should be Used?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.

If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.

Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.

The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

After you've installed the appropriate driver, it is time to establish a database connection using JDBC.

The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps –

- **Import JDBC Packages:** Add **import** statements to your Java program to import required classes in your Java code.

- **Register JDBC Driver:** This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- **Database URL Formulation:** This is to create a properly formatted address that points to the database to which you wish to connect.
- **Create Connection Object:**

Finally, code a call to the *DriverManager* object's *getConnection()* method to establish actual database connection.

Import JDBC Packages

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code –

```
import java.sql.*; // for standard JDBC programs
```

```
import java.math.*; // for BigDecimal and BigInteger support
```

Register JDBC Driver

You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces.

You need to do this registration only once in your program. You can register a driver in one of two ways.

Approach I - `Class.forName()`

The most common approach to register a driver is to use Java's **`Class.forName()`** method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses `Class.forName()` to register the Oracle driver –

```

try{

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

}

catch(ClassNotFoundException ex){

System.out.println("Error: unable to load driver class!");

System.exit(1);

}

```

You can use **getInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows –

```

try{

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

}

catch(ClassNotFoundException ex){

System.out.println("Error: unable to load driver class!");

System.exit(1);

catch(IllegalAccessException ex){

System.out.println("Error: access problem while loading!");

System.exit(2);

catch(InstantiationException ex){

System.out.println("Error: unable to instantiate driver!");

System.exit(3);

}

```

Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver –

```
try{  
  
Driver myDriver =neworacle.jdbc.driver.OracleDriver();  
  
DriverManager.registerDriver( myDriver );  
  
}  
  
catch(ClassNotFoundException ex){  
  
System.out.println("Error: unable to load driver class!");  
  
System.exit(1);  
  
}
```

Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods –

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql: //hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin: @hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2: hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds: hostname: port Number/databaseName

All the highlighted part in URL format is static and you need to change only the remaining part as per your database setup.

Create Connection Object

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object.

Using a Database URL with a username and password

The most commonly used form of **getConnection()** requires you to pass a database URL, *username*, and a *password*:

Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would be –

```
jdbc:oracle:thin:@amrood:1521:EMP
```

Now you have to call getConnection() method with appropriate username and password to get a **Connection** object as follows –

```
String URL ="jdbc:oracle:thin:@amrood:1521:EMP";
```

```
String USER ="username";
```

```
String PASS ="password"
```

```
Connection conn =DriverManager.getConnection(URL, USER, PASS);
```

Using Only a Database URL

A second form of the DriverManager.getConnection() method requires only a database URL –

```
DriverManager.getConnection(String url);
```

However, in this case, the database URL includes the username and password and has the following general form –

```
jdbc:oracle:driver:username/password@database
```

So, the above connection can be created as follows –

```
String URL ="jdbc:oracle:thin:username/password@amrood:1521:EMP";
```

```
Connection conn =DriverManager.getConnection(URL);
```

Using a Database URL and a Properties Object

A third form of the DriverManager.getConnection() method requires a database URL and a Properties object –

```
DriverManager.getConnection(String url,Properties info);
```

A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples, use the following code –

```
import java.util.*;

String URL ="jdbc:oracle:thin:@amrood:1521:EMP";

Properties info =newProperties();

info.put("user","username");

info.put("password","password");

Connection conn =DriverManager.getConnection(URL, info);
```

Closing JDBC Connections

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object.

To ensure that a connection is closed, you could provide a 'finally' block in your code. A *finally* block always executes, regardless of an exception occurs or not.

To close the above opened connection, you should call close() method as follows –

```
conn.close();
```

Explicitly closing a connection conserves DBMS resources, which will make your database administrator happy.

Once a connection is obtained we can interact with the database. The *JDBC Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use the for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

The Statement Objects

Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's `createStatement()` method, as in the following example –

```
Statement stmt = null;
```

```
try {
```

```

stmt = conn.createStatement();

...

}

catch (SQLException e) {

...

}

finally {

...

}

```

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL):** Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL):** Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL):** Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the `close()` method will do the job. If you close the `Connection` object first, it will close the `Statement` object as well. However, you should always explicitly close the `Statement` object to ensure proper cleanup.

```
Statement stmt = null;

try {

    stmt = conn.createStatement();

    ...

}

catch (SQLException e) {

    ...

}

finally {

    stmt.close();

}
```

The PreparedStatement Objects

The *PreparedStatement* interface extends the `Statement` interface, which gives you added functionality with a couple of advantages over a generic `Statement` object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object

```
PreparedStatement pstmt = null;

try {
```

```

String SQL = "Update Employees SET age = ? WHERE id = ?";

pstmt = conn.prepareStatement(SQL);

...

}

catch (SQLException e) {

...

}

finally {

...

}

```

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an **SQLException**.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database (a) **execute()**, (b) **executeQuery()**, and (c) **executeUpdate()** also work with the **PreparedStatement** object. However, the methods are modified to use SQL statements that can input the parameters.

Closing PreparedStatement Object

Just as you close a **Statement** object, for the same reason you should also close the **PreparedStatement** object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
```

```
try {
```

```
    String SQL = "Update Employees SET age = ? WHERE id = ?";
```

```
pstmt = conn.prepareStatement(SQL);
```

```
    ...
```

```
}
```

```
catch (SQLException e) {
```

```
    ...
```

```
}
```

```
finally {
```

```
pstmt.close();
```

```
}
```

The CallableStatement Objects

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object

Suppose, you need to execute the following Oracle stored procedure –

```
CREATE OR REPLACE PROCEDURE getEmpName
```

```

(EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS

BEGIN

    SELECT first INTO EMP_FIRST

    FROM Employees

    WHERE ID = EMP_ID;

END;

```

NOTE: Above stored procedure has been written for Oracle, but we are working with MySQL database so, let us write same stored procedure for MySQL as follows to create it in EMP database –

```

DELIMITER $$

DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$

CREATE PROCEDURE `EMP`.`getEmpName`

    (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))

BEGIN

    SELECT first INTO EMP_FIRST

    FROM Employees

    WHERE ID = EMP_ID;

END $$

DELIMITER ;

```

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Here are the definitions of each –

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure –

```

CallableStatement cstmt = null;

try {

    String SQL = "{call getEmpName (?, ?)}";

    cstmt = conn.prepareCall (SQL);

    ...

}

catch (SQLException e) {

    ...

}

finally {

```

```
...  
}
```

The String variable SQL, represents the stored procedure, with parameter placeholders.

Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

Closing CallableStatement Object

Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;  
  
try {  
    String SQL = "{call getEmpName (?, ?)}";  
  
    cstmt = conn.prepareCall (SQL);  
  
    ...  
}
```



```

catch (SQLException e) {

    ...

}

finally {

cstmt.close();

}

```

The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set.

The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories –

- **Navigational methods:** Used to move the cursor around.
- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.

JDBC provides the following connection methods to create statements with desired ResultSet –

- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**
- **prepareCall(String sql, int RSType, int RSConcurrency);**

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Concurrency of ResultSet

The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default

ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.
----------------------------	-----------------------------------

All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object –

```
try{

Statement stmt =conn.createStatement(

ResultSet.TYPE_FORWARD_ONLY,

ResultSet.CONCUR_READ_ONLY);

}

catch(Exception ex){

....

}

finally{

....

}
```

Navigating a Result Set

There are several methods in the ResultSet interface that involve moving the cursor, including –

S.N.	Methods & Description
1	<p>public void beforeFirst() throws SQLException</p> <p>Moves the cursor just before the first row.</p>

2	public void afterLast() throws SQLException Moves the cursor just after the last row.
3	public boolean first() throws SQLException Moves the cursor to the first row.
4	public void last() throws SQLException Moves the cursor to the last row.
5	public boolean absolute(int row) throws SQLException Moves the cursor to the specified row.
6	public boolean relative(int row) throws SQLException Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
7	public boolean previous() throws SQLException Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
8	public boolean next() throws SQLException Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
9	public int getRow() throws SQLException Returns the row number that the cursor is pointing to.

10	public void moveToInsertRow() throws SQLException Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	public void moveToCurrentRow() throws SQLException Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

Viewing a Result Set

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions –

- One that takes in a column name.
- One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet –

S.N.	Methods & Description
1	public int getInt(String columnName) throws SQLException Returns the int in the current row in the column named columnName.
2	public int getInt(int columnIndex) throws SQLException Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Similarly, there are get methods in the `ResultSet` interface for each of the eight Java primitive types, as well as common types such as `java.lang.String`, `java.lang.Object`, and `java.net.URL`.

There are also methods for getting SQL data types `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `java.sql.Clob`, and `java.sql.Blob`. Check the documentation for more information about using these SQL data types.

Updating a Result Set

The `ResultSet` interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type –

- One that takes in a column name.
- One that takes in a column index.

For example, to update a `String` column of the current row of a result set, you would use one of the following `updateString()` methods –

S.N.	Methods & Description
1	<code>public void updateString(int columnIndex, String s) throws SQLException</code> Changes the <code>String</code> in the specified column to the value of <code>s</code> .
2	<code>public void updateString(String columnName, String s) throws SQLException</code> Similar to the previous method, except that the column is specified by its name instead of its index.

There are update methods for the eight primitive data types, as well as `String`, `Object`, `URL`, and the SQL data types in the `java.sql` package.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

S.N.	Methods & Description
1	public void updateRow() Updates the current row by updating the corresponding row in the database.
2	public void deleteRow() Deletes the current row from the database
3	public void refreshRow() Refreshes the data in the result set to reflect any recent changes in the database.
4	public void cancelRowUpdates() Cancels any updates made on the current row.
5	public void insertRow() Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

- 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.

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 –

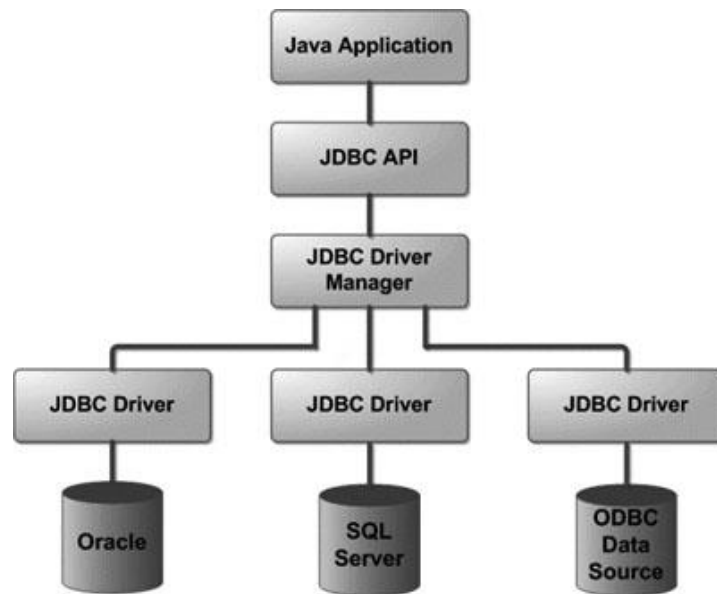


Fig: JDBC driver Manager

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.

What is JDBC Driver?

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server.

For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

The *Java.sql* package that ships with JDK, contains various classes with their behaviours defined and their actual implementations are done in third-party drivers. Third party vendors implements the *java.sql.Driver* interface in their database driver.

JDBC Drivers Types

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

Type 1: JDBC-ODBC Bridge Driver

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 2: JDBC-Native API

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3: JDBC-Net pure Java

In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

Type 4: 100% Pure Java

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

MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Which Driver should be Used?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.

If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.

Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.

The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

After you've installed the appropriate driver, it is time to establish a database connection using JDBC.

The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps –

- **Import JDBC Packages:** Add **import** statements to your Java program to import required classes in your Java code.
- **Register JDBC Driver:** This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- **Database URL Formulation:** This is to create a properly formatted address that points to the database to which you wish to connect.
- **Create Connection Object:**

Finally, code a call to the *DriverManager* object's *getConnection()* method to establish actual database connection.

Import JDBC Packages

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code –

```
import java.sql.*; // for standard JDBC programs
```

```
import java.math.*; // for BigDecimal and BigInteger support
```

Register JDBC Driver

You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces.

You need to do this registration only once in your program. You can register a driver in one of two ways.

Approach I - Class.forName()

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses Class.forName() to register the Oracle driver –

```
try{

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

}

catch(ClassNotFoundException ex){

System.out.println("Error: unable to load driver class!");

System.exit(1);

}
```

You can use **getInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows –

```
try{

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

}

catch(ClassNotFoundException ex){

System.out.println("Error: unable to load driver class!");

System.exit(1);

catch(IllegalAccessException ex){

System.out.println("Error: access problem while loading!");

System.exit(2);

catch(InstantiationException ex){

System.out.println("Error: unable to instantiate driver!");

System.exit(3);

}
```

Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver,

is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver –

```
try{
```

```

Driver myDriver =neworacle.jdbc.driver.OracleDriver();

DriverManager.registerDriver( myDriver );

}

catch(ClassNotFoundException ex){

System.out.println("Error: unable to load driver class!");

System.exit(1);

}

```

Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded **DriverManager.getConnection()** methods –

- `getConnection(String url)`
- `getConnection(String url, Properties prop)`
- `getConnection(String url, String user, String password)`

Here each form requires a database **URL**. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql:// hostname/ databaseName

ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin: @hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2: hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds: hostname: port Number/databaseName

All the highlighted part in URL format is static and you need to change only the remaining part as per your database setup.

Create Connection Object

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object.

Using a Database URL with a username and password

The most commonly used form of **getConnection()** requires you to pass a database URL, *username*, and a *password*:

Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would be

—

jdbc:oracle:thin:@amrood:1521:EMP

Now you have to call **getConnection()** method with appropriate username and password to get a **Connection** object as follows —

String URL ="jdbc:oracle:thin:@amrood:1521:EMP";


```
String USER ="username";
```

```
String PASS ="password"
```

```
Connection conn =DriverManager.getConnection(URL, USER, PASS);
```

Using Only a Database URL

A second form of the DriverManager.getConnection() method requires only a database URL –

```
DriverManager.getConnection(String url);
```

However, in this case, the database URL includes the username and password and has the following general form –

```
jdbc:oracle:driver:username/password@database
```

So, the above connection can be created as follows –

```
String URL ="jdbc:oracle:thin:username/password@amrood:1521:EMP";
```

```
Connection conn =DriverManager.getConnection(URL);
```

Using a Database URL and a Properties Object

A third form of the DriverManager.getConnection() method requires a database URL and a Properties object –

```
DriverManager.getConnection(String url,Properties info);
```

A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples, use the following code –

```
import java.util.*;
```

```
String URL ="jdbc:oracle:thin:@amrood:1521:EMP";
```

```
Properties info =newProperties();  
  
info.put("user","username");  
  
info.put("password","password");
```

```
Connection conn =DriverManager.getConnection(URL, info);
```

Closing JDBC Connections

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the `close()` method associated with connection object.

To ensure that a connection is closed, you could provide a 'finally' block in your code. *Afinally* block always executes, regardless of an exception occurs or not.

To close the above opened connection, you should call `close()` method as follows –

```
conn.close();
```

Explicitly closing a connection conserves DBMS resources, which will make your database administrator happy.

Once a connection is obtained we can interact with the database. The *JDBC Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use the for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

Servlets and JSP

Servlet is Java EE server driven technology to create web applications in java. The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing our own servlets.

All servlets must implement the javax.servlet.Servlet interface, which defines servlet lifecycle methods. When implementing a generic service, we can extend the GenericServlet class provided with the Java Servlet API. The HttpServlet class provides methods, such as doGet() and doPost(), for handling HTTP-specific services.

Most of the times, web applications are accessed using HTTP protocol and that's why we mostly extend HttpServlet class.

Common Gateway Interface (CGI)

Before introduction of Java Servlet API, CGI technology was used to create dynamic web applications. CGI technology has many drawbacks such as creating separate process for each request, platform dependent code (C, C++), high memory usage and slow performance.

CGI vs Servlet

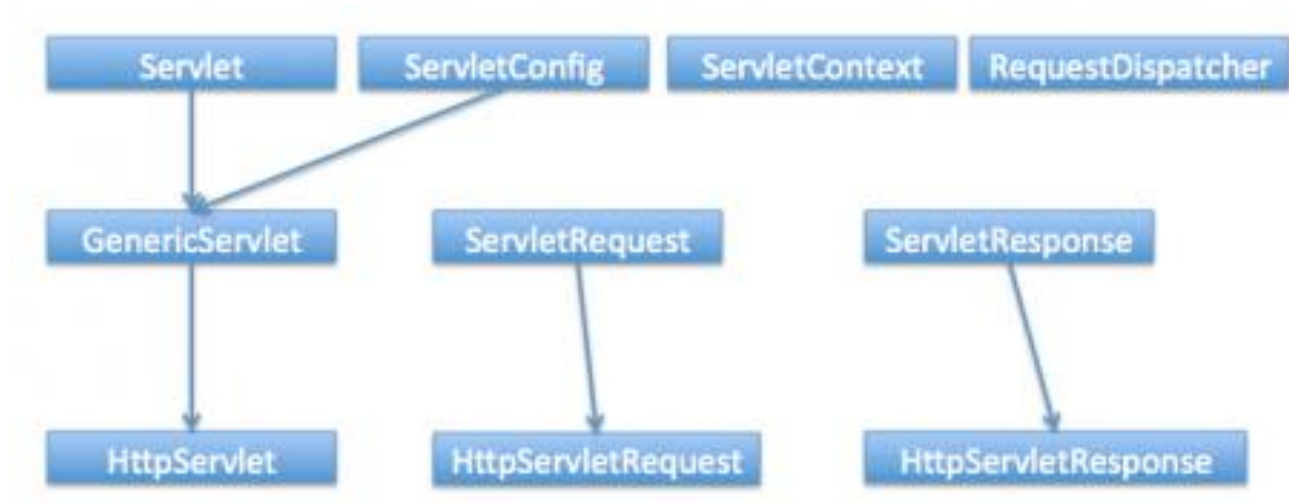
Java Servlet technology was introduced to overcome the shortcomings of CGI technology.

- Servlets provide better performance than CGI in terms of processing time, memory utilization because servlets use benefits of multithreading and for each request a new thread is created, that is faster than loading creating new Object for each request with CGI.
- Servlets are platform and system independent, the web application developed with Servlet can be run on any standard web container such as Tomcat, JBoss, Glassfish servers and on operating systems such as Windows, Linux, Unix, Solaris, Mac etc.
- Servlets are robust because container takes care of life cycle of servlet and we don't need to worry about memory leaks, security, garbage collection etc.

- Servlets are maintainable and learning curve is small because all we need to take care is business logic for our application.

Servlet API Hierarchy

`javax.servlet.Servlet` is the base **interface** of Servlet API. There are some other interfaces and classes that we should be aware of when working with Servlets. Also with Servlet 3.0 specs, servlet API introduced the use of annotations rather than having all the servlet configuration in the deployment descriptor. In this section, we will look into important Servlet API interfaces, classes and annotations that we will use further in developing our application. The below diagram shows servlet API hierarchy.



1. Servlet Interface

`javax.servlet.Servlet` is the base interface of **Java Servlet API**. Servlet interface declares the life cycle methods of servlet. All the servlet classes are required to implement this interface. The methods declared in this interface are:

1. **public abstract void init(ServletConfig paramServletConfig) throws ServletException** – This is the very important method that is invoked by servlet container to initialize the servlet and `ServletConfig` parameters. The servlet is not ready to process client request until unless `init()` method is finished executing. This method is called only once in servlet lifecycle and makes Servlet class different from normal Java objects. We can

extend this method in our servlet classes to initialize resources such as DB Connection, Socket connection etc.

2. **public abstract ServletConfig getServletConfig()** – This method returns a servlet config object, which contains any initialization parameters and startup configuration for this servlet. We can use this method to get the init parameters of servlet defines in deployment descriptor (web.xml) or through annotation in Servlet 3. We will look into ServletConfig interface later on.
3. **public abstract void service(ServletRequest req, ServletResponse res) throws ServletException, IOException** – This method is responsible for processing the client request. Whenever servlet container receives any request, it creates a new thread and execute the service() method by passing request and response as argument. Servlets usually run in multi-threaded environment, so it's developer responsibility to keep shared resources thread-safe using [synchronization](#).
4. **public abstract String getServletInfo()** – This method returns string containing information about the servlet, such as its author, version, and copyright. The string returned should be plain text and can't have markups.
5. **public abstract void destroy()** – This method can be called only once in servlet life cycle and used to close any open resources. This is like finalize method of a java class.

2. ServletConfig Interface

javax.servlet.ServletConfig is used to pass configuration information to Servlet. Every servlet has it's own ServletConfig object and servlet container is responsible for instantiating this object. We can provide servlet init parameters in **web.xml** file or through use of WebInitParam annotation. We can use **getServletConfig()** method to get the ServletConfig object of the servlet.

The important methods of ServletConfig interface are:

1. **public abstract ServletContext getServletContext()** – This method returns the ServletContext object for the servlet. We will look into ServletContext interface in next section.

2. **public abstract Enumeration<String> getInitParameterNames()** – This method returns the Enumeration<String> of name of init parameters defined for the servlet. If there are no init parameters defined, this method returns empty enumeration.
3. **public abstract String getInitParameter(String paramString)** – This method can be used to get the specific init parameter value by name. If parameter is not present with the name, it returns null.

3. ServletContext interface

javax.servlet.ServletContext interface provides access to web application variables to the servlet. The ServletContext is unique object and available to all the servlets in the web application. When we want some init parameters to be available to multiple or all of the servlets in the web application, we can use ServletContext object and define parameters in web.xml using **<context-param>** element. We can get the ServletContext object via the `getServletContext()` method of ServletConfig. Servlet engines may also provide context objects that are unique to a group of servlets and which is tied to a specific portion of the URL path namespace of the host.

Some of the important methods of ServletContext are:

1. **public abstract ServletContext getContext(String uripath)** – This method returns ServletContext object for a particular uripath or null if not available or not visible to the servlet.
2. **public abstract URL getResource(String path) throws MalformedURLException** – This method return URL object allowing access to any content resource requested. We can access items whether they reside on the local file system, a remote file system, a database, or a remote network site without knowing the specific details of how to obtain the resources.
3. **public abstract InputStream getResourceAsStream(String path)** – This method returns an input stream to the given resource path or null if not found.
4. **public abstract RequestDispatcher getRequestDispatcher(String urlpath)** – This method is mostly used to obtain a reference to another servlet. After obtaining a

RequestDispatcher, the servlet programmer forward a request to the target component or include content from it.

5. **public abstract void log(String msg)** – This method is used to write given message string to the servlet log file.
6. **public abstract Object getAttribute(String name)** – Return the object attribute for the given name. We can get enumeration of all the attributes using **public abstract Enumeration<String> getAttributeNames()** method.
7. **public abstract void setAttribute(String paramString, Object paramObject)** – This method is used to set the attribute with application scope. The attribute will be accessible to all the other servlets having access to this ServletContext. We can remove an attribute using **public abstract void removeAttribute(String paramString)** method.
8. **String getInitParameter(String name)** – This method returns the String value for the init parameter defined with name in web.xml, returns null if parameter name doesn't exist. We can use **Enumeration<String> getInitParameterNames()** to get enumeration of all the init parameter names.
9. **boolean setInitParameter(String paramString1, String paramString2)** – We can use this method to set init parameters to the application.

Note: Ideally the name of this interface should be ApplicationContext because it's for the application and not specific to any servlet. Also don't get confused it with the servlet context passed in the URL to access the web application.

4. ServletRequest interface

ServletRequest interface is used to provide client request information to the servlet. Servlet container creates ServletRequest object from client request and pass it to the servlet service() method for processing.

Some of the important methods of ServletRequest interface are:

1. **Object getAttribute(String name)** – This method returns the value of named attribute as Object and null if it's not present. We can use `getAttributeNames()` method to get the enumeration of attribute names for the request. This interface also provide methods for setting and removing attributes.
2. **String getParameter(String name)** – This method returns the request parameter as String. We can use `getParameterNames()` method to get the enumeration of parameter names for the request.
3. **String getServerName()** – returns the hostname of the server.
4. **int getServerPort()** – returns the port number of the server on which it's listening.

The child interface of `ServletRequest` is `HttpServletRequest` that contains some other methods for session management, cookies and authorization of request.

5. `ServletResponse` interface

`ServletResponse` interface is used by servlet in sending response to the client. Servlet container creates the `ServletResponse` object and pass it to `servlet service()` method and later use the response object to generate the HTML response for client.

Some of the important methods in `HttpServletResponse` are:

1. **void addCookie(Cookie cookie)** – Used to add cookie to the response.
2. **void addHeader(String name, String value)** – used to add a response header with the given name and value.
3. **String encodeURL(java.lang.String url)** – encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.
4. **String getHeader(String name)** – return the value for the specified header, or null if this header has not been set.
5. **void sendRedirect(String location)** – used to send a temporary redirect response to the client using the specified redirect location URL.

6. **void setStatus(int sc)** – used to set the status code for the response.

6. RequestDispatcher interface

RequestDispatcher interface is used to forward the request to another resource that can be HTML, JSP or another servlet in the same context. We can also use this to include the content of another resource to the response. This interface is used for servlet communication within the same context.

There are two methods defined in this interface:

1. **void forward(ServletRequest request, ServletResponse response)** – forwards the request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
2. **void include(ServletRequest request, ServletResponse response)** – includes the content of a resource (servlet, JSP page, HTML file) in the response.

We can get RequestDispatcher in a servlet using ServletContext *getRequestDispatcher(String path)* method. The path must begin with a / and is interpreted as relative to the current context root.

7. GenericServlet class

GenericServlet is an **abstract class** that implements Servlet, ServletConfig and Serializable interface. GenericServlet provide default implementation of all the Servlet life cycle methods and ServletConfig methods and makes our life easier when we extend this class, we need to override only the methods we want and rest of them we can work with the default implementation. Most of the methods defined in this class are only for easy access to common methods defined in Servlet and ServletConfig interfaces.

One of the important method in GenericServlet class is no-argument init() method and we should override this method in our servlet program if we have to initialize some resources before processing any request from servlet.

8. HTTPServlet class

HTTPServlet is an abstract class that extends GenericServlet and provides the base for creating HTTP based web applications. There are methods defined to be overridden by subclasses for different HTTP methods.

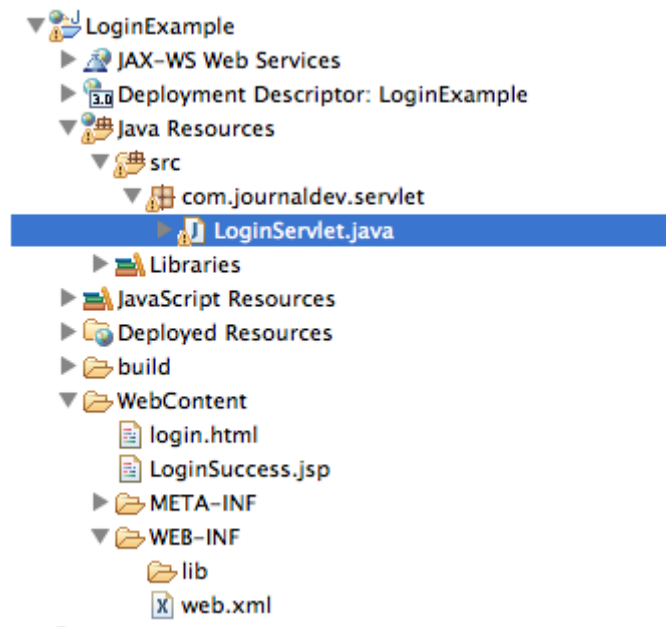
1. doGet(), for HTTP GET requests

2. doPost(), for HTTP POST requests
3. doPut(), for HTTP PUT requests
4. doDelete(), for HTTP DELETE requests

Java Servlet Login Example

To create our login servlet example, we will use simple HTML, JSP, and servlet that will authenticate the user credentials. We will also see the use of ServletContext init parameters, attributes, ServletConfig init parameters and RequestDispatcher include() and response sendRedirect() usage.

Our Final Dynamic Web Project will look like below image. I am using Eclipse and Tomcat for the application, the process to create dynamic web project is provided in Java Web Applications tutorial.



Here is our login HTML page, we will put it in the welcome files list in the web.xml so that when we launch the application it will open the login page.

```
<!DOCTYPE html>

<html>

<head>

<meta charset="US-ASCII">

<title>Login Page</title>

</head>

<body>

<form action="LoginServlet" method="post">

Username: <input type="text" name="user">

<br>

Password: <input type="password" name="pwd">

<br>

<input type="submit" value="Login">

</form>

</body>

</html>
```

If the login will be successful, the user will be presented with new JSP page with login successful message. JSP page code is like below.

```
<% @ page language="java" contentType="text/html; charset=US-ASCII"
```

```

    pageEncoding="US-ASCII"%>

<!DOCTYPE    html    PUBLIC    "-//W3C//DTD    HTML    4.01    Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">

<title>Login Success Page</title>

</head>

<body>

<h3>Hi Pankaj, Login successful.</h3>

<a href="login.html">Login Page</a>

</body>

</html>

```

Here is the web.xml deployment descriptor file where we have defined servlet context init parameters and welcome page.

```

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">

    <display-name>LoginExample</display-name>

    <welcome-file-list>

        <welcome-file>login.html</welcome-file>

```

```
</welcome-file-list>
```

```
<context-param>
```

```
<param-name>dbURL</param-name>
```

```
<param-value>jdbc:mysql://localhost/mysql_db</param-value>
```

```
</context-param>
```

```
<context-param>
```

```
<param-name>dbUser</param-name>
```

```
<param-value>mysql_user</param-value>
```

```
</context-param>
```

```
<context-param>
```

```
<param-name>dbUserPwd</param-name>
```

```
<param-value>mysql_pwd</param-value>
```

```
</context-param>
```

```
</web-app>
```

Here is our final Servlet class for authenticating the user credentials, notice the use of annotations for Servlet configuration and ServletConfig init parameters.

```
package com.journaldev.servlet;
```

```
import java.io.IOException;
```

```
import java.io.PrintWriter;
```

```
import javax.servlet.RequestDispatcher;
```

```

import javax.servlet.ServletException;

import javax.servlet.annotation.WebInitParam;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

/**

 * Servlet Tutorial - Servlet Example

 */

@WebServlet(

    description = "Login Servlet",

    urlPatterns = { "/LoginServlet" },

    initParams = {

        @WebInitParam(name = "user", value = "Pankaj"),

        @WebInitParam(name = "password", value = "journaldev")

    })

public class LoginServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public void init() throws ServletException {

        //we can create DB connection resource here and set it to Servlet context

        if(getServletContext().getInitParameter("dbURL").equals("jdbc:mysql://localhost/mysql_db") &&

```

```

        getServletContext().getInitParameter("dbUser").equals("mysql_user") &&
        getServletContext().getInitParameter("dbUserPwd").equals("mysql_pwd"))

getServletContext().setAttribute("DB_Success", "True");

else throw new ServletException("DB Connection error");

}

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

    //get request parameters for userID and password

    String user = request.getParameter("user");

    String pwd = request.getParameter("pwd");

    //get servlet config init params

    String userID = getServletConfig().getInitParameter("user");

    String password = getServletConfig().getInitParameter("password");

    //logging example

    log("User="+user+"::password="+pwd);

    if(userID.equals(user) && password.equals(pwd)){

        response.sendRedirect("LoginSuccess.jsp");

    }else{

        RequestDispatcher rd = getServletContext().getRequestDispatcher("/login.html");

        PrintWriter out= response.getWriter();

        out.println("<font color=red>Either user name or password is wrong.</font>");

        rd.include(request, response);

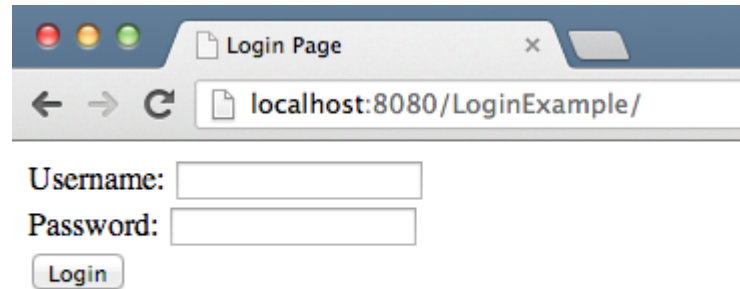
```


}

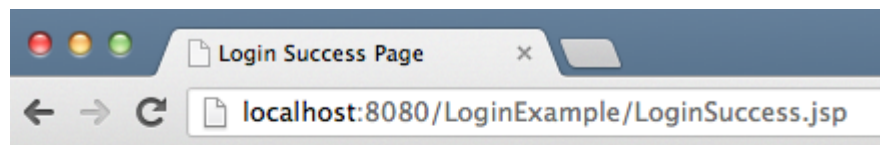
}

}

Below screenshots shows the different pages of our Servlet Example project, based on the user password combinations for successful login and failed logins.



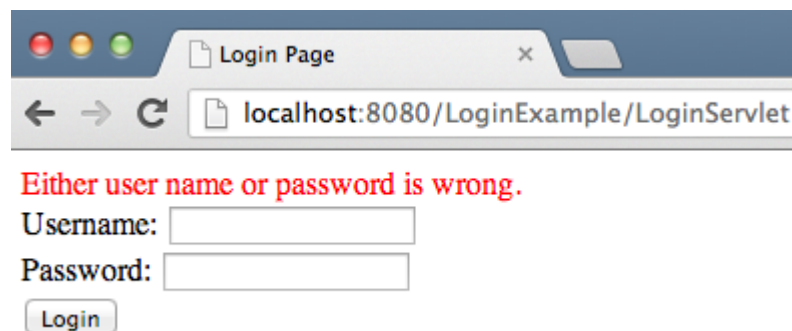
A screenshot of a web browser window titled "Login Page". The address bar shows "localhost:8080/LoginExample/". The page contains a form with two input fields: "Username:" and "Password:". Below the "Password:" field is a "Login" button.



A screenshot of a web browser window titled "Login Success Page". The address bar shows "localhost:8080/LoginExample/LoginSuccess.jsp".

Hi Pankaj, Login successful.

[Login Page](#)



A screenshot of a web browser window titled "Login Page". The address bar shows "localhost:8080/LoginExample/LoginServlet". The page displays a red error message: "Either user name or password is wrong." Below the message is a form with "Username:" and "Password:" input fields and a "Login" button.

JSP

What is JSP and why do we need JSP?

JSP (JavaServer Pages) is server side technology to create dynamic java web application. JSP can be thought as an extension to servlet technology because it provides features to easily create user views.

JSP Page consists of HTML code and provide option to include java code for dynamic content. Since web applications contain a lot of user screens, JSPs are used a lot in web applications. To bridge the gap between java code and HTML in JSP, it provides additional features such as JSP Tags, Expression Language, Custom tags. This makes it easy to understand and helps a web developer to quickly develop JSP pages.

Advantages of JSP over Servlets?

- We can generate HTML response from servlets also but the process is cumbersome and error prone, when it comes to writing a complex HTML response, writing in a servlet will be a nightmare. JSP helps in this situation and provide us flexibility to write normal HTML page and include our java code only where it's required.
- JSP provides additional features such as tag libraries, expression language, custom tags that helps in faster development of user views.
- JSP pages are easy to deploy, we just need to replace the modified page in the server and container takes care of the deployment. For servlets, we need to recompile and deploy whole project again.

Actually Servlet and JSPs compliment each other. We should use Servlet as server side controller and to communicate with model classes whereas JSPs should be used for presentation layer.

Life cycle of JSP Page

JSP life cycle is also managed by container. Usually every web container that contains servlet container also contains JSP container for managing JSP pages.

JSP pages life cycle phases are:

- **Translation** – JSP pages doesn't look like normal java classes, actually JSP container parse the JSP pages and translate them to generate corresponding servlet source code. If JSP file name is home.jsp, usually its named as home_jsp.java.
- **Compilation** – If the translation is successful, then container compiles the generated servlet source file to generate class file.
- **Class Loading** – Once JSP is compiled as servlet class, its lifecycle is similar to servlet and it gets loaded into memory.
- **Instance Creation** – After JSP class is loaded into memory, its object is instantiated by the container.
- **Initialization** – The JSP class is then initialized and it transforms from a normal class to servlet. After initialization, ServletConfig and ServletContext objects become accessible to JSP class.
- **Request Processing** – For every client request, a new thread is spawned with ServletRequest and ServletResponse to process and generate the HTML response.
- **Destroy** – Last phase of JSP life cycle where it's unloaded into memory.

Life cycle methods of JSP

JSP lifecycle methods are:

jspInit() declared in JspPage interface. This method is called only once in JSP lifecycle to initialize config params.

_jspService(HttpServletRequest request, HttpServletResponse response) declared in HttpJspPage interface and response for handling client requests.

jspDestroy() declared in JspPage interface to unload the JSP from memory.

Simple JSP Example with Eclipse and Tomcat

We can use Eclipse IDE for building dynamic web project with JSPs and use Tomcat to run it. Please read Java Web Applications tutorial to learn how can we easily create JSPs in Eclipse and run it in tomcat.

A simple JSP example page example is:

home.jsp

```
<% @ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">

<title>First JSP</title>

</head>

<% @ page import="java.util.Date" %>

<body>

<h3>Hi Pankaj</h3><br>

<strong>Current Time is</strong>: <%=new Date() %>

</body>

</html>
```

If you have a simple JSP that uses only JRE classes, we are not required to put it as WAR file. Just create a directory in the tomcat webapps folder and place your JSP file in the newly created directory. For

example, if your JSP is located at apache-tomcat/webapps/test/home.jsp, then you can access it in browser with URL `http://localhost:8080/test/home.jsp`. If your host and port is different, then you need to make changes in URL accordingly.

JSP Files location in Web Application WAR File

We can place JSP files at any location in the WAR file, however if we put it inside the WEB-INF directory, we won't be able to access it directly from client.

We can configure JSP just like servlets in web.xml, for example if I have a JSP example page like below inside WEB-INF directory:

test.jsp

```
<% @ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">

<title>Test JSP</title>

</head>

<body>

Test JSP Page inside WEB-INF folder.<br>

Init Param "test" value =<%=config.getInitParameter("test") %><br>

HashCode of this object=<%=this.hashCode() %>

</body>
```

</html>

And configure it in web.xml configuration as:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
```

```
  <display-name>FirstJSP</display-name>
```

```
  <servlet>
```

```
    <servlet-name>Test</servlet-name>
```

```
    <jsp-file>/WEB-INF/test.jsp</jsp-file>
```

```
  <init-param>
```

```
    <param-name>test</param-name>
```

```
    <param-value>Test Value</param-value>
```

```
  </init-param>
```

```
  </servlet>
```

```
  <servlet-mapping>
```

```
    <servlet-name>Test</servlet-name>
```

```
    <url-pattern>/Test.do</url-pattern>
```

```
  </servlet-mapping>
```

```
  <servlet>
```

```
    <servlet-name>Test1</servlet-name>
```

```
<jsp-file>/WEB-INF/test.jsp</jsp-file>
```

```
</servlet>
```

```
<servlet-mapping>
```

```
<servlet-name>Test1</servlet-name>
```

```
<url-pattern>/Test1.do</url-pattern>
```

```
</servlet-mapping>
```

```
</web-app>
```

Then we can access it with both the URLs <http://localhost:8080/FirstJSP/Test.do> and <http://localhost:8080/FirstJSP/Test1.do>

Notice that container will create two instances in this case and both will have their own servlet config objects, you can confirm this by visiting these URLs in browser.

For Test.do URI, you will get response like below.

Test JSP Page inside WEB-INF folder.

Init Param "test" value =Test Value

HashCode of this object=1839060256

For Test1.do URI, you will get response like below.

Test JSP Page inside WEB-INF folder.

Init Param "test" value =null

HashCode of this object=38139054

Notice the init param value in second case is null because it's not defined for the second servlet, also notice the hashCode is different.

If you will make further requests, the hashCode value will not change because the requests are processed by spawning a new thread by the container.

Did you noticed the use of **config** variable in above JSP example but there is no variable declared, it's because its one of the 9 implicit objects available in JSP page, read more about them at **JSP Implicit Objects**.

JSP API Interfaces and Classes

All the core JSP interfaces and classes are defined in javax.servlet.jsp package. Expression Language API interfaces are classes are part of javax.servlet.jsp.el package. JSP Tag Libraries interfaces and classes are defined in javax.servlet.jsp.tagext package.

Here we will look into interfaces and classes of Core JSP API.

JspPage Interface

JspPage interface extends Servlet interface and declares jspInit() and jspDestroy() life cycle methods of the JSP pages.

HttpJspPage Interface

HttpJspPage interface describes the interaction that a JSP Page Implementation Class must satisfy when using the HTTP protocol. This interface declares the service method of JSP page for HTTP protocol as *public void _jspService(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException*.

JspWriter abstract Class

Similar to PrintWriter in servlets with additional facility of buffering support. This is one of the implicit variables in a JSP page with name "out". This class extends java.io.Writer and container provide their own implementation for this abstract class and use it while translating JSP page to Servlet. We can get it's

object using `PageContext.getOut()` method. Apache Tomcat concrete class for `JspWriter` is `org.apache.jasper.runtime.JspWriterImpl`.

JspContext abstract Class

`JspContext` serves as the base class for the `PageContext` class and abstracts all information that is not specific to servlets. The `JspContext` provides mechanism to obtain the `JspWriter` for output, mechanism to work with attributes and API to manage the various scoped namespaces.

PageContext abstract Class

`PageContext` extends `JspContext` to provide useful context information when JSP is used for web applications. A `PageContext` instance provides access to all the namespaces associated with a JSP page, provides access to several page attributes, as well as a layer above the implementation details. Implicit objects are added to the `pageContext` automatically.

JspFactory abstract Class

The `JspFactory` is an abstract class that defines a number of factory methods available to a JSP page at runtime for the purposes of creating instances of various interfaces and classes used to support the JSP implementation.

JspEngineInfo abstract Class

The `JspEngineInfo` is an abstract class that provides information on the current JSP engine.

ErrorData final Class

Contains information about an error, for error pages.

JspException Class

A generic exception known to the JSP container, similar to `ServletException`. If JSP pages throw `JspException` then errorpage mechanism is used to present error information to user.

JspTagException Class

Exception to be used by a Tag Handler to indicate some unrecoverable error.

SkipPageException Class

Exception to indicate the calling page must cease evaluation. Thrown by a simple tag handler to indicate that the remainder of the page must not be evaluated. This exception should not be thrown manually in a JSP page.

JSP Comments

Since JSP is built on top of HTML, we can write comments in JSP file like html comments as

```
<-- This is HTML Comment -->
```

These comments are sent to the client and we can look it with view source option of browsers.

We can put comments in JSP files as:

```
<%-- This is JSP Comment--%>
```

This comment is suitable for developers to provide code level comments because these are not sent in the client response.

JSP Scriptlets

Scriptlet tags are the easiest way to put java code in a JSP page. A scriptlet tag starts with <% and ends with %>.

Any code written inside the scriptlet tags go into the `_jspService()` method.

For example:

```
<%
```

```
    Date d = new Date();
```

```
    System.out.println("Current Date="+d);
```

```
%>
```

JSP Expression

Since most of the times we print dynamic data in JSP page using *out.print()* method, there is a shortcut to do this through JSP Expressions. JSP Expression starts with `<%=` and ends with `%>`.

`<% out.print("Pankaj"); %>` can be written using JSP Expression as `<%= "Pankaj" %>`

Notice that anything between `<%= %>` is sent as parameter to *out.print()* method. Also notice that scriptlets can contain multiple java statements and always ends with semicolon (;) but expression doesn't end with semicolon.

JSP Directives

JSP Directives are used to give special instructions to the container while JSP page is getting translated to servlet source code. JSP directives starts with `<%@` and ends with `%>`

For example, in above JSP Example, I am using *page* directive to to instruct container JSP translator to import the Date class.

JSP Declaration

JSP Declarations are used to declare member methods and variables of servlet class. JSP Declarations starts with `<%!` and ends with `%>`.

For example we can create an int variable in JSP at class level as `<%! public static int count=0; %>`

JSP transformed Servlet Source Code and Class File location in Tomcat

Once JSP files are translated to Servlet source code, the source code (.java) and compiled classes both are place in **Tomcat/work/Catalina/localhost/FirstJSP/org/apache/jsp** directory. If the JSP files are inside other directories of application, the directory structure is maintained.

For JSPs inside WEB-INF directory, its source and class files are inside **Tomcat/work/Catalina/localhost/FirstJSP/org/apache/jsp/WEB_002dINF** directory.

Here is the source code generated for above test.jsp page.

test_jsp.java

```

/*

* Generated by the Jasper component of Apache Tomcat

* Version: Apache Tomcat/7.0.32

*/

package org.apache.jsp.WEB_002dINF;

import javax.servlet.*;

import javax.servlet.http.*;

import javax.servlet.jsp.*;

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

    private static final javax.servlet.jsp.JspFactory _jspxFactory =
        javax.servlet.jsp.JspFactory.getDefaultFactory();

    private static java.util.Map<java.lang.String,java.lang.Long> _jspx_dependants;

    private javax.el.ExpressionFactory _el_expressionfactory;

    private org.apache.tomcat.InstanceManager _jsp_instancemanager;

    public java.util.Map<java.lang.String,java.lang.Long> getDependants() {

```

```

    return _jspx_dependants;

}

public void _jspInit() {

    _el_expressionfactory =
    _jspxFactory.getJspApplicationContext(getServletConfig().getServletContext()).getExpressionFactory();

    _jsp_instancemanager =
    org.apache.jasper.runtime.InstanceManagerFactory.getInstanceManager(getServletConfig());

}

public void _jspDestroy() {

}

public void _jspService(final javax.servlet.http.HttpServletRequest request, final
javax.servlet.http.HttpServletResponse response)

    throws java.io.IOException, javax.servlet.ServletException {

    final javax.servlet.jsp.PageContext pageContext;

    javax.servlet.http.HttpSession session = null;

    final javax.servlet.ServletContext application;

    final javax.servlet.ServletConfig config;

    javax.servlet.jsp.JspWriter out = null;

    final java.lang.Object page = this;

```

```

javax.servlet.jsp.JspWriter _jspx_out = null;

javax.servlet.jsp.PageContext _jspx_page_context = null;


try {

    response.setContentType("text/html; charset=US-ASCII");

    pageContext = _jspxFactory.getPageContext(this, request, response,

                                                null, true, 8192, true);

    _jspx_page_context = pageContext;

    application = pageContext.getServletContext();

    config = pageContext.getServletConfig();

    session = pageContext.getSession();

    out = pageContext.getOut();

    _jspx_out = out;

    out.write("\n");

    out.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\"
\"http://www.w3.org/TR/html4/loose.dtd\">\n");

    out.write("<html>\n");

    out.write("<head>\n");

    out.write("<meta http-equiv=\"Content-Type\" content=\"text/html; charset=US-ASCII\">\n");

    out.write("<title>Test JSP</title>\n");

    out.write("</head>\n");

```

```

out.write("<body>\n");

out.write("Test JSP Page inside WEB-INF folder.<br>\n");

out.write("Init Param \"test\" value =");

out.print(config.getInitParameter("test") );

out.write("<br>\n");

out.write("HashCode of this object=");

out.print(this.hashCode() );

out.write("\n");

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

out.write("</html>");

} catch (java.lang.Throwable t) {

    if (!(t instanceof javax.servlet.jsp.SkipPageException)){

        out = _jspx_out;

        if (out != null && out.getBufferSize() != 0)

            try { out.clearBuffer(); } catch (java.io.IOException e) {}

        if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);

        else throw new ServletException(t);

    }

} finally {

    _jspxFactory.releasePageContext(_jspx_page_context);

}

```

```
}  
  
}
```

Notice following points in above servlet code;

- The package of class starts with org.apache.jsp and if JSPs are inside other folders, it includes directory hierarchy too. Usually we don't care about it.
- The generated servlet class is final and can't be extended.
- It extends org.apache.jasper.runtime.HttpJspBase that is similar to HttpServlet except that it's internal to Tomcat JSP Translator implementation. HttpJspBase extends HttpServlet and implements HttpJspPage interface.
- Notice the local variables at the start of _jspService() method implementation, they are automatically added by JSP translator and available for use in service methods, i.e in scriptlets.

As a java programmer, sometimes it helps to look into the generated source for debugging purposes.

JSP init parameters

We can define init parameters for the JSP page as shown in above example and we can retrieve them in JSP using **config** implicit object, we will look into implicit objects in JSP in more detail in future posts.

Overriding JSP init() method

We can override JSP init method for creating resources to be used by JSP service() method using JSP Declaration tags, we can override jspInit() and jspDestroy() or any other methods also. However we should never override _jspService() method because anything we write in JSP goes into service method.

Attributes in a JSP

Apart from standard servlet attributes with request, session and context scope, in JSP we have another scope for attributes, i.e Page Scope that we can get from pageContext object. We will look its importance in custom tags tutorial. For normal JSP programming, we don't need to worry about page scope.

XML and Web Services

XML is a software- and hardware-independent tool for storing and transporting data.

What is XML?

- XML stands for eXtensible Markup Language
- XML is a markup language much like HTML
- XML was designed to store and transport data
- XML was designed to be self-descriptive
- XML is a W3C Recommendation

XML Does Not DO Anything

Maybe it is a little hard to understand, but XML does not DO anything.

This note is a note to Tove from Jani, stored as XML:

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

The XML above is quite self-descriptive:

- It has sender information.
- It has receiver information
- It has a heading
- It has a message body.

But still, the XML above does not DO anything. XML is just information wrapped in tags.

Someone must write a piece of software to send, receive, store, or display it:

Note

To: Tove

From: Jani

Reminder

Don't forget me this weekend!

The Difference Between XML and HTML

XML and HTML were designed with different goals:

- XML was designed to carry data - with focus on what data is
- HTML was designed to display data - with focus on how data looks
- XML tags are not predefined like HTML tags are

Important contents beyond syllabus

Struts 2

The **struts 2 framework** is used to develop **MVC-based web application**.

The struts framework was initially created by **Craig McClanahan** and donated to Apache Foundation in May, 2000 and Struts 1.0 was released in June 2001.

The current stable release of Struts is Struts 2.3.16.1 in March 2, 2014.

This struts 2 tutorial covers all the topics of Struts 2 Framework with simplified examples for beginners and experienced persons.

Struts 2 Framework

The Struts 2 framework is used to develop MVC (Model View Controller) based web applications. Struts 2 is the combination of **webwork framework** of opensymphony and **struts 1**.

$$\text{struts2} = \text{webwork} + \text{struts1}$$

The Struts 2 provides supports to POJO based actions, Validation Support, AJAX Support, Integration support to various frameworks such as Hibernate, Spring, Tiles etc, support to various result types such as Freemarker, Velocity, JSP etc.

Struts2 Features

Struts 2 provides many features that were not in struts 1. The **important features** of struts 2 framework are as follows:

1. Configurable MVC components
2. POJO based actions
3. AJAX support
4. Integration support
5. Various Result Types

6. Various Tag support

7. Theme and Template support

1) Configurable MVC components

In struts 2 framework, we provide all the components (view components and action) information in struts.xml file. If we need to change any information, we can simply change it in the xml file.

2) POJO based actions

In struts 2, action class is POJO (Plain Old Java Object) i.e. a simple java class. Here, you are not forced to implement any interface or inherit any class.

3) AJAX support

Struts 2 provides support to ajax technology. It is used to make asynchronous request i.e. it doesn't block the user. It sends only required field data to the server side not all. So it makes the performance fast.

4) Integration Support

We can simply integrate the struts 2 application with hibernate, spring, tiles etc. frameworks.

5) Various Result Types

We can use JSP, freemarker, velocity etc. technologies as the result in struts 2.

6) Various Tag support

Struts 2 provides various types of tags such as UI tags, Data tags, control tags etc to ease the development of struts 2 application.

7) Theme and Template support

Struts 2 provides three types of theme support: xhtml, simple and css_xhtml. The xhtml is default theme of struts 2. Themes and templates can be used for common look and feel.

Model1 vs Model2

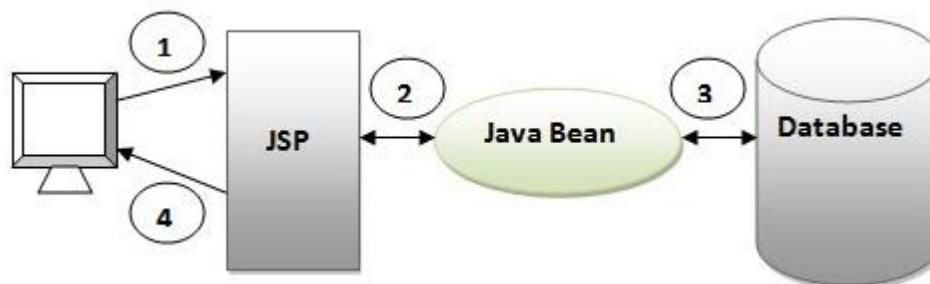
Model 1 Architecture

Servlet and JSP are the main technologies to develop the web applications.

Servlet was considered superior to CGI. Servlet technology doesn't create process, rather it creates thread to handle request. The advantage of creating thread over process is that it doesn't allocate separate memory area. Thus many subsequent requests can be easily handled by servlet.

Problem in Servlet technology Servlet needs to recompile if any designing code is modified. It doesn't provide separation of concern. Presentation and Business logic are mixed up.

JSP overcomes almost all the problems of Servlet. It provides better separation of concern, now presentation and business logic can be easily separated. You don't need to redeploy the application if JSP page is modified. JSP provides support to develop web application using JavaBean, custom tags and JSTL so that we can put the business logic separate from our JSP that will be easier to test and debug.



As you can see in the above figure, there is picture which shows the flow of the model1 architecture.

1. Browser sends request for the JSP page
2. JSP accesses Java Bean and invokes business logic
3. Java Bean connects to the database and get/save data
4. Response is sent to the browser which is generated by JSP

Advantage of Model 1 Architecture

- Easy and Quick to develop web application

Disadvantage of Model 1 Architecture

- **Navigation control is decentralized** since every page contains the logic to determine the next page. If JSP page name is changed that is referred by other pages, we need to change it in all the pages that leads to the maintenance problem.
- **Time consuming** You need to spend more time to develop custom tags in JSP. So that we don't need to use scriptlet tag.
- **Hard to extend** It is better for small applications but not for large applications.

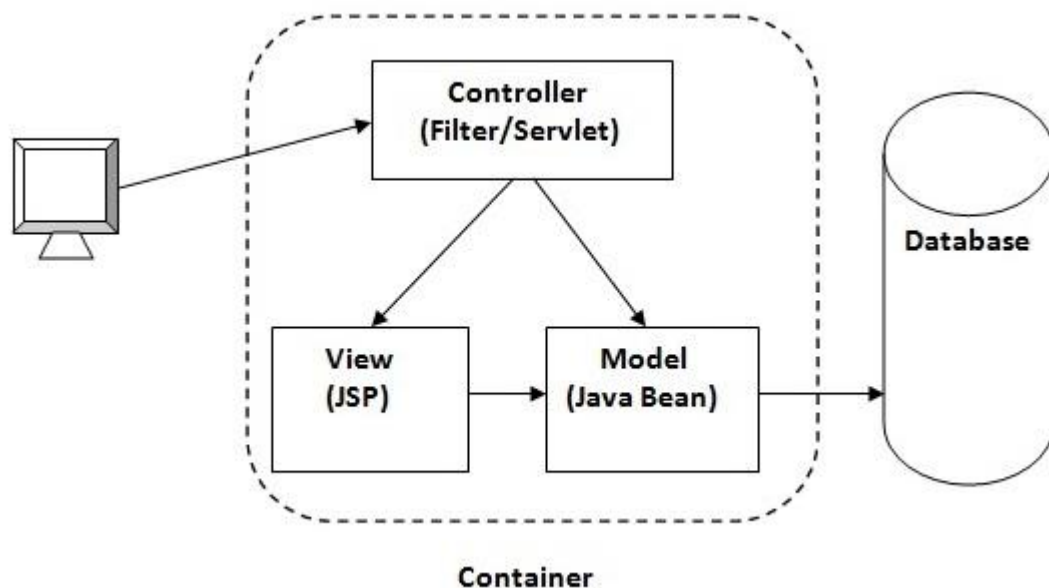
Model 2 (MVC) Architecture

Model 2 is based on the MVC (Model View Controller) design pattern. The MVC design pattern consists of three modules model, view and controller.

Model The model represents the state (data) and business logic of the application.

View The view module is responsible to display data i.e. it represents the presentation.

Controller The controller module acts as an interface between view and model. It intercepts all the requests i.e. receives input and commands to Model / View to change accordingly.



Advantage of Model 2 (MVC) Architecture

- **Navigation control is centralized** Now only controller contains the logic to determine the next page.
- **Easy to maintain**
- **Easy to extend**
- **Easy to test**
- **Better separation of concerns**

Disadvantage of Model 2 (MVC) Architecture

- We need to write the controller code self. If we change the controller code, we need to recompile the class and redeploy the application.

Struts 2 Architecture

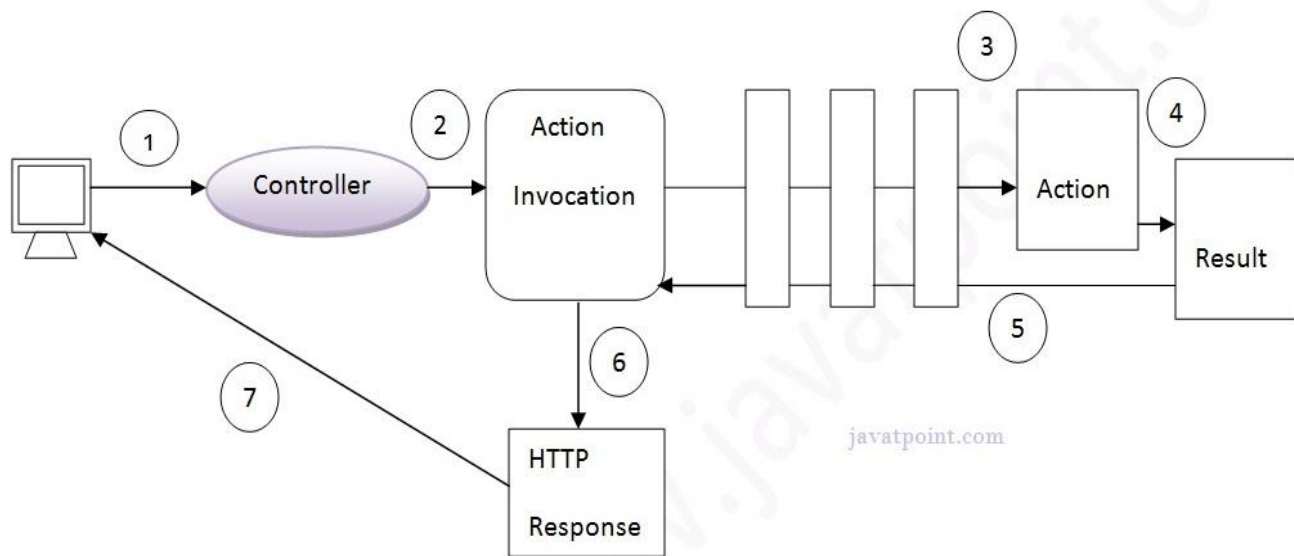
The **architecture and flow of struts 2 application**, is combined with many components such as Controller, ActionProxy, ActionMapper, Configuration Manager, ActionInvocation, Inerceptor, Action, Result etc.

Here, we are going to understand the struts flow by 2 ways:

1. struts 2 basic flow
2. struts 2 standard architecture and flow provided by apache struts

Struts 2 basic flow

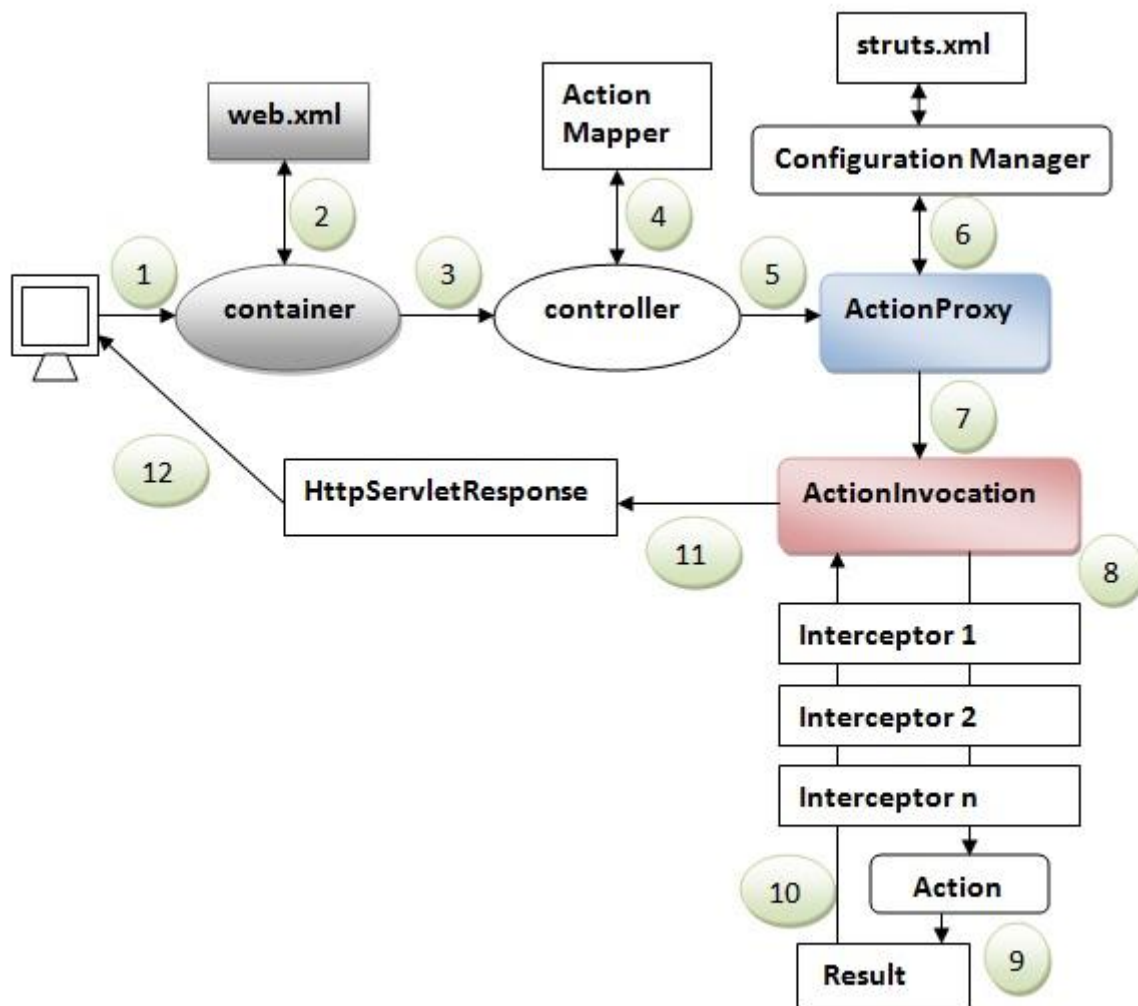
Let's try to understand the basic flow of struts 2 application by this simple figure:



1. User sends a request for the action
2. Controller invokes the ActionInvocation
3. ActionInvocation invokes each interceptors and action
4. A result is generated
5. The result is sent back to the ActionInvocation
6. A HttpServletResponse is generated
7. Response is sent to the user

Struts 2 standard flow (Struts 2 architecture)

Let's try to understand the standard architecture of struts 2 application by this simple figure:



1. User sends a request for the action
2. Container maps the request in the web.xml file and gets the class name of controller.
3. Container invokes the controller (StrutsPrepareAndExecuteFilter or FilterDispatcher). Since struts2.1, it is StrutsPrepareAndExecuteFilter. Before 2.1 it was FilterDispatcher.
4. Controller gets the information for the action from the ActionMapper
5. Controller invokes the ActionProxy
6. ActionProxy gets the information of action and interceptor stack from the configuration manager which gets the information from the struts.xml file.

7. ActionProxy forwards the request to the ActionInvocation
8. ActionInvocation invokes each interceptors and action
9. A result is generated
10. The result is sent back to the ActionInvocation
11. A HttpServletResponse is generated
12. Response is sent to the user

JavaMail

The JavaMail is an API that is used to compose, write and read electronic messages (emails).

The JavaMail API provides protocol-independent and platform-independent framework for sending and receiving mails.

The javax.mail and javax.mail.activation packages contain the core classes of JavaMail API.

The JavaMail facility can be applied to many events. It can be used at the time of registering the user (sending notification such as thanks for your interest to my site), forgot password (sending password to the users email id), sending notifications for important updates etc. So there can be various usage of java mail api.

Protocols used in JavaMail API

There are some protocols that are used in JavaMail API.

- SMTP
- POP
- IMAP
- MIME

- NNTP and others

SMTP

SMTP is an acronym for Simple Mail Transfer Protocol. It provides a mechanism to deliver the email. We can use Apache James server, Postcast server, cmail server etc. as an SMTP server. But if we purchase the host space, an SMTP server is by default provided by the host provider. For example, my smtp server is mail.javatpoint.com. If we use the SMTP server provided by the host provider, authentication is required for sending and receiving emails.

POP

POP is an acronym for Post Office Protocol, also known as POP3. It provides a mechanism to receive the email. It provides support for single mail box for each user. We can use Apache James server, cmail server etc. as an POP server. But if we purchase the host space, an POP server is by default provided by the host provider. For example, the pop server provided by the host provider for my site is mail.javatpoint.com. This protocol is defined in RFC 1939.

IMAP

IMAP is an acronym for Internet Message Access Protocol. IMAP is an advanced protocol for receiving messages. It provides support for multiple mail box for each user ,in addition to, mailbox can be shared by multiple users. It is defined in RFC 2060.

MIME

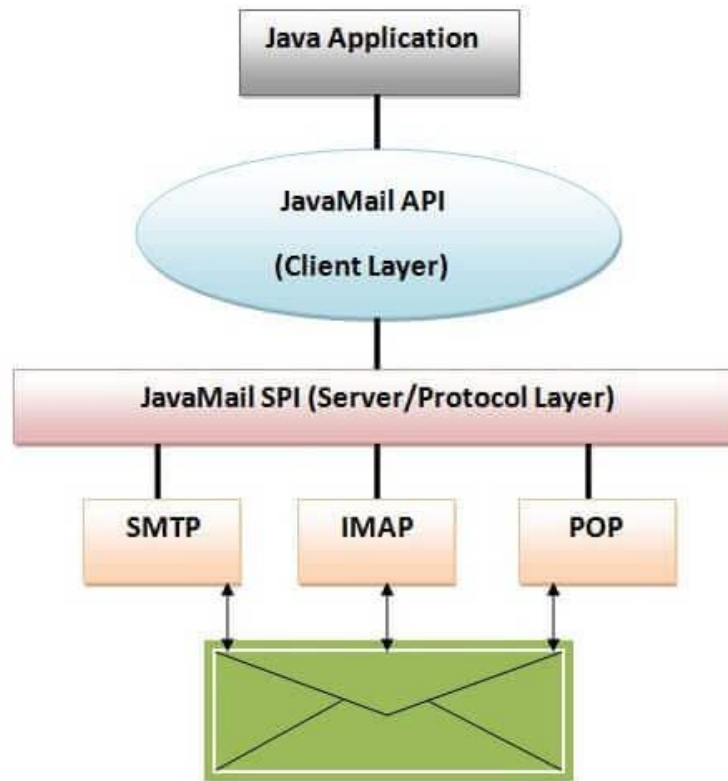
Multiple Internet Mail Extension (MIME) tells the browser what is being sent e.g. attachment, format of the messages etc. It is not known as mail transfer protocol but it is used by your mail program.

NNTP and Others

There are many protocols that are provided by third-party providers. Some of them are Network News Transfer Protocol (NNTP), Secure Multipurpose Internet Mail Extensions (S/MIME) etc.

JavaMail Architecture

The java application uses JavaMail API to compose, send and receive emails. The JavaMail API uses SPI (Service Provider Interfaces) that provides the intermediary services to the java application to deal with the different protocols. Let's understand it with the figure given below:



JavaMail API Core Classes

There are two packages that are used in Java Mail API: `javax.mail` and `javax.mail.internet` package. These packages contain many classes for Java Mail API. They are:

- `javax.mail.Session` class
- `javax.mail.Message` class
- `javax.mail.internet.MimeMessage` class
- `javax.mail.Address` class
- `javax.mail.internet.InternetAddress` class

- javax.mail.Authenticator class
- javax.mail.PasswordAuthentication class
- javax.mail.Transport class
- javax.mail.Store class
- javax.mail.Folder class etc

Hibernate

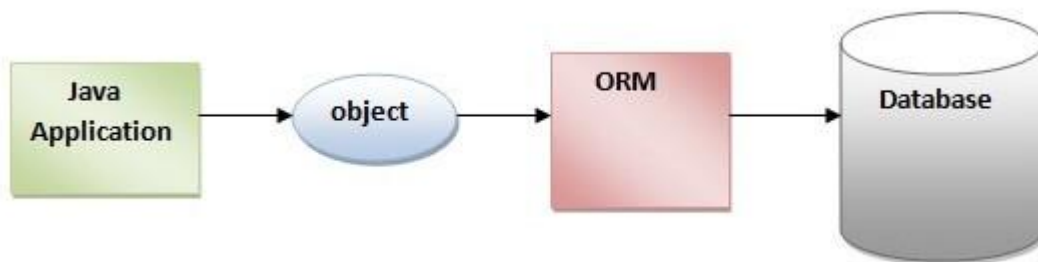
This hibernate tutorial provides in-depth concepts of Hibernate Framework with simplified examples. It was started in 2001 by Gavin King as an alternative to EJB2 style entity bean.

Hibernate Framework

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

ORM Tool

An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.



The ORM tool internally uses the JDBC API to interact with the database.

What is JPA?

Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools. The **javax.persistence** package contains the JPA classes and interfaces.

Advantages of Hibernate Framework

Following are the advantages of hibernate framework:

1) Open Source and Lightweight

Hibernate framework is open source under the LGPL license and lightweight.

2) Fast Performance

The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled by default.

3) Database Independent Query

HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, if database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.

4) Automatic Table Creation

Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

5) Simplifies Complex Join

Fetching data from multiple tables is easy in hibernate framework.

6) Provides Query Statistics and Database Status

Hibernate supports Query cache and provide statistics about query and database status.

Spring

This spring tutorial provides in-depth concepts of Spring Framework with simplified examples. It was developed by Rod Johnson in 2003. Spring framework makes the easy development of JavaEE application.

It is helpful for beginners and experienced persons.

Spring Framework

Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc. We will learn these modules in next page. Let's understand the IOC and Dependency Injection first.

Inversion Of Control (IOC) and Dependency Injection

These are the design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain. Let's understand this with the following code:

```
class Employee{  
  
    Address address;  
  
    Employee(){  
  
        address=new Address();  
  
    }  
  
}
```

In such case, there is dependency between the Employee and Address (tight coupling). In the Inversion of Control scenario, we do this something like this:

```
class Employee{
```

```
Address address;
```

```
Employee(Address address){
```

```
    this.address=address;
```

```
}
```

```
}
```

Thus, IOC makes the code loosely coupled. In such case, there is no need to modify the code if our logic is moved to new environment.

In Spring framework, IOC container is responsible to inject the dependency. We provide metadata to the IOC container either by XML file or annotation.

Advantage of Dependency Injection

- makes the code loosely coupled so easy to maintain
- makes the code easy to test

Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development

The Dependency Injection feature of Spring Framework and its support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

References

- [1]. Herbert Schildt, Java : The Complete Reference, 9th Edition, Oracle Press.
- [2]. Gary Cornell, Core Java Volume II Advanced Features, 8th Edition, Pearson Education.
- [3]. Jim Keogh, J2ee : Complete Reference, 1st Edition, Tata McGraw Hill.
- [4]. James Gosling, Ken Arnold and David Holmes, Java Programming Language, 4th Edition, Pearson Education.
- [5]. Gary Cornell, Core Java Volume I, 3rd Edition, Pearson Education.
- [6]. <https://www.javatpoint.com/java-tutorial>
- [7]. <https://www.geeksforgeeks.org/java/>
- [8]. <https://www.tutorialspoint.com/java/index.htm>
- [9]. *Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad; Buckley, Alex (2014). The Java® Language Specification (PDF) (Java SE 8 ed.).*
- [10]. *Gosling, James; Joy, Bill; Steele, Guy L., Jr.; Bracha, Gilad (2005). The Java Language Specification (3rd ed.). Addison-Wesley. ISBN 0-321-24678-0.*
- [11]. *Lindholm, Tim; Yellin, Frank (1999). The Java Virtual Machine Specification (2nd ed.). Addison-Wesley. ISBN 0-201-43294-3.*