**JAVA8 FEATURES**

**Functional Interfaces**

An Interface that contains exactly one abstract method is known as functional interface.It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

Example:-

1. @FunctionalInterface
2. interface sayable{
3. void say(String msg); // abstract method
4. int hashCode();
5. String toString();
6. boolean equals(Object obj);
7. }
8. public class FunctionalInterfaceExample2 implements sayable{
9. public void say(String msg){
10. System.out.println(msg);
11. }
12. public static void main(String[] args) {
13. FunctionalInterfaceExample2 fie = new FunctionalInterfaceExample2();
14. fie.say("Hello there");
15. }
16. }

Output:

Hellothere

Functional interfaces are ddivided into four types mainly:-

•Predicate

•Consumer

•Supplier

•Function

**Predicate**

A predicate functional interface of java is a type of function which accepts a single value or argument and does some sort of processing on it, and returns a boolean (True/ False) answer. The implementation of the Predicate functional interface also encapsulates the logic of filtering in Java. Predicate functional interface also has some extensions. These are IntPredicate, DoublePredicate, and LongPredicate. These types of predicate functional interfaces accept only primitive data types or values as arguments.

Bi-Predicate- takes two arguments, does some processing, and returns the boolean value.

**Syntax:-**

public interface Predicate{

boolean test(T t);

}

Example:-

import java.util.function.Predicate;

public class PredicateInterfaceExample {

public static void main(String[] args) {

Predicate<Integer> pr = a -> (a > 18); // Creating predicate

System.out.println(pr.test(10)); // Calling Predicate method

}

}

Output:

False

#### **Consumer**

Consumer interface accepts only one argument or a gentrified argument. It has no return value. There are also functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer. These variants accept primitive values as arguments. Bi-Consumer interface takes two arguments.

**Syntax :-**

Consumer<Integer> consumer = (value) -> System.out.println(value);

Example:

**i**mport java.util.function.Consumer;

public class ConsumerInterfaceExample {

static void printMessage(String name){

System.out.println("Hello "+name);

}

static void printValue(int val){

System.out.println(val);

}

public static void main(String[] args) {

Consumer<String> consumer1 = ConsumerInterfaceExample::printMessage;

consumer1.accept("John");

Consumer<Integer> consumer2 = ConsumerInterfaceExample::printValue;

consumer2.accept(12);

}

}

Output:

Hello John

12

**Function**

A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing.

**syntax of Bi-Function –**

@FunctionalInterface  
public interface BiFunction<T, U, R>   
{  
   
 R apply(T t, U u);  
 .......  
   
}

T, U are the inputs, and there is only one output that is R.

Example:

import java.util.function.Function;

public class FunctionInterfaceExample {

static String show(String message){

return "Hello "+message;

}

public static void main(String[] args) {

Function<String, String> fun = FunctionInterfaceExample::show; System.out.println(fun.apply("Peter"));

}

}

Output:

Hello Peter

**Supplier**

It represents a function which does not take in any argument but produces a value of type T.The lambda expression assigned to an object of Supplier type is used to define its get() which eventually produces a value. Suppliers are useful when we don’t need to supply any value and obtain a result at the same time.

Example:

|  |
| --- |
| **import** java.util.function.Supplier;    **public** **class** Main {  **public** **static** **void** main(String args[])  {      Supplier<Double> randomValue = () -> Math.random();    System.out.println(randomValue.get());  }  } |

**Output:**0.5685808855697841

**Lambda Expressions**

It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.It treated as a function, so compiler does not create .class file.

## **Syntax**

(argument-list) -> {body}

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

Example:

interface Drawable{

public void draw();

}

public class LambdaExpressionExample2 {

public static void main(String[] args) {

int width=10;

//with lambda

Drawable d2=()->{

System.out.println("Drawing "+width);

};

d2.draw();

}

}

Output:

Drawing 10

**Method References**

A lambda expression only calls an existing method. In those cases, it looks clear to refer to the existing method by name. The method references can do this, they are compact, easy-to-read as compared to lambda expressions. A method reference is the shorthand syntax for a lambda expression that contains just one method call.

**Generic syntax:** Method reference

**A.** To refer to a method in an object

Object :: methodName

**B.** To print all elements in a list

Following is an illustration of a lambda expression that just calls a single method in its entire execution:

list.forEach(s -> System.out.println(s));

**C.** Shorthand to print all elements in a list

To make the code clear and compact, In the above example, one can turn lambda expression into a method reference:

list.forEach(System.out::println);

**Stream API**

Stream provides following features:

* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Stream is lazy and evaluates code only when required.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc.

Operations On Streams:-

•**map:** The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

•**filter:** The filter method is used to select elements as per the Predicate passed as argument.

•**sorted:** The sorted method is used to sort the stream.

Terminal Operations:

•**collect:** The collect method is used to return the result of the intermediate operations performed on the stream.

•**forEach**: The forEach method is used to iterate through every element of the stream.

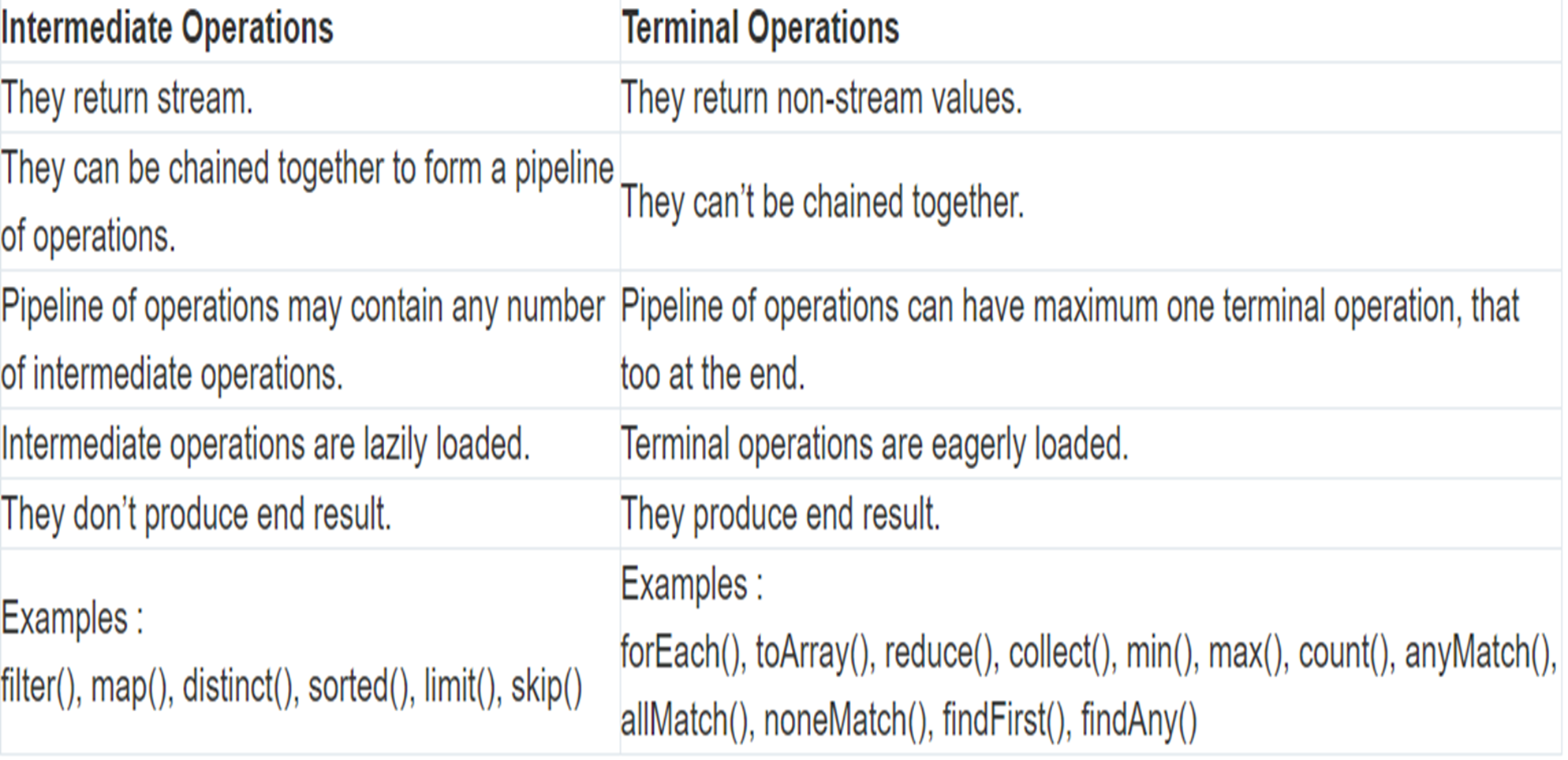
•**reduce:** The reduce method is used to reduce the elements of a stream to a single value.The reduce method takes a BinaryOperator as a parameter.

Example:-

|  |
| --- |
| import java.util.\*;  import java.util.stream.\*;    class Demo  {  public static void main(String args[])  {    List<Integer> number = Arrays.asList(2,3,4,5);    List<Integer> square = number.stream().map(x -> x\*x).  collect(Collectors.toList());  System.out.println(square);    List<String> names =  Arrays.asList("Reflection","Collection","Stream");    List<String> result = names.stream().filter(s->s.startsWith("S")).  collect(Collectors.toList());  System.out.println(result);    List<String> show =names.stream().sorted().collect(Collectors.toList());  System.out.println(show);    List<Integer> numbers = Arrays.asList(2,3,4,5,2);  Set<Integer> squareSet =numbers.stream().map(x->x\*x).collect(Collectors.toSet());  System.out.println(squareSet);    number.stream().map(x->x\*x).forEach(y->System.out.println(y));  int even =number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);  System.out.println(even);  }  } |

Output:

[4, 9, 16, 25]  
[Stream]  
[Collection, Reflection, Stream]  
[16, 4, 9, 25]  
4  
9  
16  
25  
6



**METHOD REFERENCES**

A method reference is the shorthand syntax for a lambda expression that contains just one method call.

**Generic syntax:** Method reference

**A.** To refer to a method in an object

Object :: methodName

**B.** To print all elements in a list

Following is an illustration of a lambda expression that just calls a single method in its entire execution:

list.forEach(s -> System.out.println(s));

**C.** Shorthand to print all elements in a list

To make the code clear and compact, In the above example, one can turn lambda expression into a method reference:

list.forEach(System.out::println);

Example:-

@FunctionalInterface   
interface Display {   
 void display();   
}   
public class Example {   
 public void myMethod() {   
 System.out.println("method reference in java 8");   
 }   
 public static void main(String[] args) {   
 Example obj = new Example();   
 // Reference to the method using the object of the class myMethod  
 Display ref = obj::myMethod;   
 // Calling the method inside the functional interface Display   
 ref.display();   
 }   
}

**OPTIONAL CLASS**

Every Java Programmer is familiar with [NullPointerException](https://www.geeksforgeeks.org/null-pointer-exception-in-java/). It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in **java.util package**. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

Example:-

**import** java.util.Optional;

**public** **class** OptionalDemo {

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

{

String[] words = **new** String[10];

Optional<String> checkNull

= Optional.ofNullable(words[5]);

**if** (checkNull.isPresent()) {

String word = words[5].toLowerCase();

System.out.print(word);

}

**else**

System.out.println("word is null");

}

}

**DEFAULT METHODS**

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

Example:-

public class Java8Tester {  
  
 public static void main(String args[]) {  
 Vehicle vehicle = new Car();  
 vehicle.print();  
 }  
}  
  
interface Vehicle {  
  
 default void print() {  
 System.out.println("I am a vehicle!");  
 }  
   
 static void blowHorn() {  
 System.out.println("Blowing horn!!!");  
 }  
}  
  
interface FourWheeler {  
  
 default void print() {  
 System.out.println("I am a four wheeler!");  
 }  
}  
  
class Car implements Vehicle, FourWheeler {  
  
 public void print() {  
 Vehicle.super.print();  
 FourWheeler.super.print();  
 Vehicle.blowHorn();  
 System.out.println("I am a car!");  
 }  
}

**DATE and TIME**

New date-time API is introduced in Java 8 to overcome the following drawbacks of old date-time API :

* **Not thread safe :** Unlike old java.util.Date which is not thread safe the new date-time API is *immutable* and doesn’t have setter methods.
* **Less operations :** In old API there are only few date operations but the new API provides us with many date operations.

Java 8 under the package java.time introduced a new date-time API, most important classes among them are :

**1)Local :** Simplified date-time API with no complexity of timezone handling.

1. **Zoned :** Specialized date-time API to deal with various timezones.

* **LocalDate/LocatTime** and **LocalDateTime API :** Use it when time zones are NOT required.
* **Zoned date-time API** : Use it when time zones are to be considered

**STRING OPERATIONS**

•**join()**: This method concatenates a sequence of strings using a specified delimiter

•**chars()**: This method returns a stream of characters from the string.

•**codePoints()**: This method returns a stream of Unicode code points from the string.

•**repeat()**: This method repeats the string a specified number of times

•**isBlank()**: This method returns **true** if a string is empty or contains only whitespace characters, and **false** otherwise.

•**strip()**: This method returns a new string with leading and trailing whitespace removed. It is similar to the **trim()** method, but also removes non-breaking space characters.

•**lines()**: This method returns a stream of lines extracted from the string, separated by line terminators.

•**formatted()**: This method returns a formatted string using the specified format string and arguments.

**HASHMAP**

* **HashMap** class is a part of the Java Collections Framework and is used to store and retrieve key-value pairs.
* It is a widely used data structure for efficient lookup, insertion, and deletion operations.
* When a value is retrieved from the **HashMap** using its key, the key is hashed to determine the index in the array, and the corresponding value is returned

*Syntax*

HashMap<KeyType, ValueType> map = new HashMap<>();

**HASHMAP CHANGES**

* Improved performance: Java 8 introduces several performance improvements to **HashMap**, including faster iteration, reduced memory consumption, and reduced collisions.​
* Default methods: **Map** interface in Java 8 introduces several default methods, which **HashMap** implements, that provide additional functionality such as **forEach**, **computeIfAbsent**, **computeIfPresent**, and **merge**.​
* Lambda expressions: Java 8 introduces lambda expressions, which can be used with **HashMap** to simplify code and provide a more concise syntax for performing operations on key-value pairs.​
* Method references: Java 8 also introduces method references, which can be used with **HashMap** to refer to a method that can be used to compute a value based on a key, rather than using a lambda expression.​
* Stream API: Java 8 introduces the **Stream** API, which can be used with **HashMap** to perform various operations on key-value pairs, such as filtering, mapping, and reducing.​
* Tree-based structure - When a **HashMap** contains a large number of key-value pairs in a single bucket, the bucket is now represented by a tree structure instead of a linked list. This improves the performance of operations like **put**, **get**, and **remove** in such cases from O(n) to O(log n).​

**METASPACE**

* In Java 8, the **Metaspace** is a part of the memory used by the JVM to store metadata about the classes that are loaded. The Metaspace is not part of the Java heap space and is allocated outside of it. ​
* Automatic memory management - The Metaspace is not a fixed-size region of memory like the PermGen space. Instead, it can expand dynamically to accommodate the metadata of loaded classes. The amount of memory used by the Metaspace can be controlled using the **-XX:MaxMetaspaceSize** command-line option.​
* Garbage collection - The Metaspace is not explicitly garbage-collected like the Java heap space. Instead, it relies on the Java garbage collector to reclaim memory when it is no longer needed. This means that memory leaks caused by class loading and unloading are less likely to occur.​
* Metadata storage - In Java 8, the metadata about loaded classes is stored in native memory instead of Java heap memory. This improves performance by reducing the number of Java heap allocations required.​
* Class unloading - Classes that are no longer used can be unloaded from the Metaspace by the garbage collector, which frees up memory. This is different from the PermGen space, where classes were never unloaded unless the JVM was restarted.​
* Out of Memory Errors - If the Metaspace runs out of memory, the JVM will automatically resize it, or if the maximum Metaspace size has been reached, it will throw an **OutOfMemoryError**.​

**MAP VS FLATMAP**

| **map()** | **flatMap()** |
| --- | --- |
| The function passed to map() operation returns a single value for a single input. | The function you pass to flatmap() operation returns an arbitrary number of values as the output. |
| One-to-one mapping occurs in map(). | One-to-many mapping occurs in flatMap(). |
| Only perform the mapping. | Perform mapping as well as flattening. |
| Produce a stream of value. | Produce a stream of stream value. |
| map() is used only for transformation. | flatMap() is used both for transformation and mapping. |

**Syntax of map() method**​

<R> Stream<R> map(Function<? **super** T,? **extends** R> mapper) ​

**Example:**​

List citylist = Arrays.asList("delhi", "mumbai", "hyderabad"). stream(). map(String::toUpperCase).collect(Collectors.toList());​

​

**Syntax of flatMap() method**​

<R> Stream<R> flatMap(Function<? **super** T,? **extends** Stream<? **extends** R>> mapper) ​

**Example:**​

List country = Stream.of(Arrays.asList( "Finland", "Greece", "Iceland",  "Mali", ), Arrays.asList("Singapore", "Turkey","Greece", "Iceland")).flatMap(List::stream) .collect(Collectors.toList()); ​

**MySQL**

MySQL is a relational database management system (RDBMS) based on the SQL (Structured Query Language) queries. It is one of the most popular languages for accessing and managing the records in the table. MySQL is open-source and free software under the GNU license. Oracle Company supports it.

Features of MySQL are:

**Relational Database Management System (RDBMS)**

[MySQL](https://www.javatpoint.com/mysql-tutorial) is a relational database management system. This database language is based on the [SQL](https://www.javatpoint.com/sql-tutorial) queries to access and manage the records of the table.

**Easy to use**

MySQL is easy to use. We have to get only the basic knowledge of SQL. We can build and interact with MySQL by using only a few simple SQL statements.

**It is secure**

MySQL consists of a solid data security layer that protects sensitive data from intruders. Also, passwords are encrypted in MySQL.

**Client/ Server Architecture**

MySQL follows the working of a client/server architecture. There is a database server (MySQL) and arbitrarily many clients (application programs), which communicate with the server; that is, they can query data, save changes, etc.

**Free to download**

MySQL is free to use so that we can download it from MySQL official website without any cost.

**It is scalable**

MySQL supports multi-threading that makes it easily scalable. It can handle almost any amount of data, up to as much as 50 million rows or more. The default file size limit is about 4 GB. However, we can increase this number to a theoretical limit of 8 TB of data.

**Speed**

MySQL is considered one of the very fast database languages, backed by a large number of the benchmark test.

**Compatible on many operating systems**

MySQL is compatible to run on many operating systems, like Novell NetWare, Windows\* Linux\*, many varieties of UNIX\* (such as Sun\* Solaris\*, AIX, and DEC\* UNIX), OS/2, FreeBSD\*, and others. MySQL also provides a facility that the clients can run on the same computer as the server or on another computer (communication via a local network or the Internet).

**Allows roll-back**

MySQL allows transactions to be rolled back, commit, and crash recovery.

**Memory efficiency**

Its efficiency is high because it has a very low memory leakage problem.

**High Performance**

MySQL is faster, more reliable, and cheaper because of its unique storage engine architecture. It provides very high-performance results in comparison to other databases without losing an essential functionality of the software. It has fast loading utilities because of the different cache memory.

# MySQL Create Database

A database is used to store the collection of records in an organized form. It allows us to hold the data into tables, rows, columns, and indexes to find the relevant information frequently. We can access and manage the records through the database very easily.

**Syntax:**

CREATE DATABASE database\_name;

**Example:**

Here, we are going to create a database name **"employeedb"** using the following statement:

**CREATE** **DATABASE** employeesdb;

# MySQL SELECT Database

SELECT Database is used in MySQL to select a particular database to work with. This query is used when multiple databases are available with MySQL Server.

**Syntax:**

USE database\_name;

**Example:**

Let's take an example to use a database name "customers".

USE customers;

# MySQL DROP Database

We can drop/delete/remove a MySQL database quickly with the MySQL DROP DATABASE command.

**Syntax:**

**DROP** **DATABASE** database\_name;

**Example:**

Suppose we want to remove a database named **"mytestdb\_copy"**. Execute the below statement:

**DROP** **DATABASE** mytestdb\_copy;

# MySQL CREATE TABLE

[MySQL](https://www.javatpoint.com/mysql-tutorial) allows us to create a table into the database by using the [**CREATE TABLE**](https://www.javatpoint.com/mysql-create-table) command

**Syntax:**

**CREATE** **TABLE** table\_name(

column\_definition1,

column\_definition2,

........

);

Example:

Here, we are going to create a table name **"employee\_table"** in the database **"employeedb"** using the following statement:

**CREATE** **TABLE** employee\_table(

id **int** NOT NULL AUTO\_INCREMENT,

**name** **varchar**(45) NOT NULL,

occupation **varchar**(35) NOT NULL,

age **int** NOT NULL,

**PRIMARY** **KEY** (id)

);

MySQL ALTER Table

MySQL ALTER statement is used when you want to change the name of your table or any table field. It is also used to add or delete an existing column in a table.

***Syntax:***

ALTER TABLE *table\_name*

ADD *column\_name datatype;*

***Example:***

ALTER TABLE *Customers*

ADD *Email Varchar(255);*

# MySQL INSERT Statement

MySQL INSERT statement is used to store or add data in MySQL table within the database

**Syntax:**

**INSERT** **INTO** table\_name ( field1, field2,...fieldN )

**VALUES**

( value1, value2,...valueN );

**Example:**

**INSERT** **INTO** People **VALUES**

(102, 'Joseph', 'Developer', 30),

(103, 'Mike', 'Leader', 28),

(104, 'Stephen', 'Scientist', 45);

# MySQL UPDATE Query

MySQL UPDATE query is a DML statement used to modify the data of the MySQL table within the database.

**Syntax:**

**UPDATE** table\_name

**SET** column\_name1 = new-value1,

column\_name2=new-value2, ...

[**WHERE** Clause];

**Example:**

**UPDATE** trainer

**SET** email = '[mike@tutorialandexamples.com](mailto:mike@tutorialandexamples.com)'

**WHERE** course\_name = 'Java';

# MySQL DELETE Statement

MySQL DELETE statement is used to remove records from the MySQL table that is no longer required in the database

**Syntax:**

**DELETE** **FROM** table\_name **WHERE** condition;

**Example:**

**DELETE** **FROM** Employees **WHERE** emp\_id=107;

MySQL Select/Use Query:

Uses:

To retrieve records of a table from a database.

Syntax: To select records from a table.

SELECT \* from table\_name;

Example:

SELECT \* from items;

# SQL JOIN

As the name shows, JOIN means *to combine something*. In case of SQL, JOIN means **"to combine two or more tables"**.

The SQL JOIN clause takes records from two or more tables in a database and combines it together.

**ANSI standard SQL** defines five types of JOIN :

1. inner join,
2. left outer join,
3. right outer join,
4. full outer join, and
5. cross join.

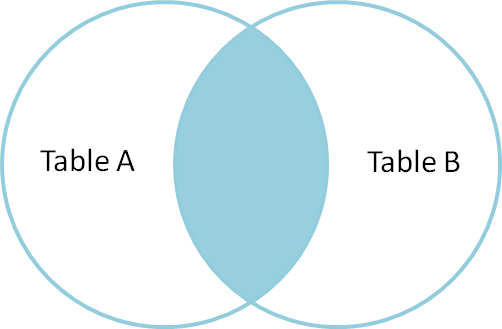
In the process of joining, rows of both tables are combined in a single table.

### **A. INNER JOIN**

The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

**Syntax**:

SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1   
INNER JOIN table2  
ON table1.matching\_column = table2.matching\_column;  
  
  
table1: First table.  
table2: Second table  
matching\_column: Column common to both the tables.



**Example Queries(INNER JOIN)**

This query will show the names and age of students enrolled in different courses.

SELECT StudentCourse.COURSE\_ID, Student.NAME, Student.AGE FROM Student  
INNER JOIN StudentCourse  
ON Student.ROLL\_NO = StudentCourse.ROLL\_NO;

**Output**:



### **B. LEFT JOIN**

This join returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1   
LEFT JOIN table2  
ON table1.matching\_column = table2.matching\_column;  
  
  
table1: First table.  
table2: Second table  
matching\_column: Column common to both the tables.

***Note****: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.*



**Example Queries(LEFT JOIN)**:

SELECT Student.NAME,StudentCourse.COURSE\_ID   
FROM Student  
LEFT JOIN StudentCourse   
ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output**:



### **C. RIGHT JOIN**

RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. For the rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1   
RIGHT JOIN table2  
ON table1.matching\_column = table2.matching\_column;  
  
  
table1: First table.  
table2: Second table  
matching\_column: Column common to both the tables.

***Note****: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.*



**Example Queries(RIGHT JOIN)**:

SELECT Student.NAME,StudentCourse.COURSE\_ID   
FROM Student  
RIGHT JOIN StudentCourse   
ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output:**



**SQL | WHERE Clause**

WHERE keyword is used for fetching **filtered data** in a result set.

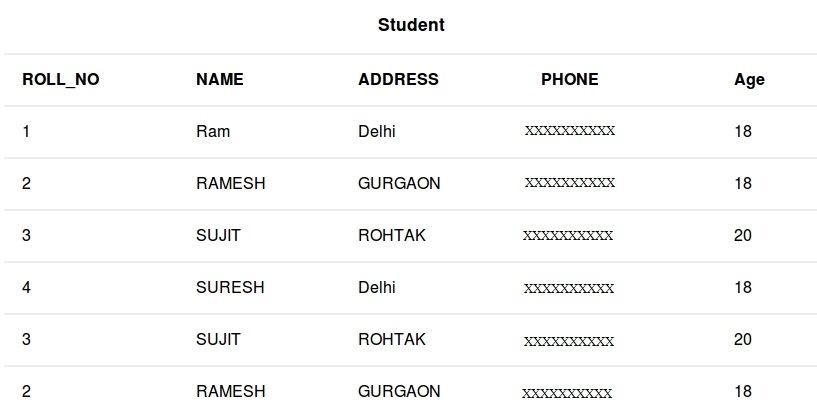
* It is used to fetch data according to a particular criteria.
* WHERE keyword can also be used to filter data by matching patterns.

**Basic Syntax:** **SELECT column1,column2 FROM table\_name WHERE column\_name operator value;**

**column1 , column2:** fields int the table  
table\_name: name of table  
column\_name: name of field used for filtering the data  
operator: operation to be considered for filtering  
value: exact value or pattern to get related data in result

**List of operators that can be used with where clause:**

|  |  |
| --- | --- |
| **operator** | **description** |
| > | Greater Than |
| >= | Greater than or Equal to |
| < | Less Than |
| <= | Less than or Equal to |
| = | Equal to |
| <> | Not Equal to |
| BETWEEN | In an inclusive Range |
| LIKE | Search for a pattern |
| IN | To specify multiple possible values for a column |



**Queries**

* To fetch record of students with age equal to 20

SELECT \* FROM Student WHERE Age=20;

* Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **Age** |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |

* To fetch Name and Address of students with ROLL\_NO greater than 3

SELECT ROLL\_NO,NAME,ADDRESS FROM Student WHERE ROLL\_NO > 3;

* Output:

|  |  |  |
| --- | --- | --- |
| **ROLL\_NO** | **NAME** | **ADDRESS** |
| 4 | SURESH | Delhi |

**BETWEEN** operator

It is used to fetch filtered data in a given range inclusive of two values. **Basic Syntax:** **SELECT column1,column2 FROM table\_name WHERE column\_name BETWEEN value1 AND value2;**

**BETWEEN:** operator name

**value1 AND value2:** exact value from value1 to value2 to get related data in result set.

**Queries**

* To fetch records of students where ROLL\_NO is between 1 and 3 (inclusive)

SELECT \* FROM Student WHERE ROLL\_NO BETWEEN 1 AND 3;

* Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **Age** |
| 1 | Ram | Delhi | XXXXXXXXXX | 18 |
| 2 | RAMESH | GURGAON | XXXXXXXXXX | 18 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 2 | RAMESH | GURGAON | XXXXXXXXXX | 18 |

* To fetch NAME,ADDRESS of students where Age is between 20 and 30 (inclusive)

SELECT NAME,ADDRESS FROM Student WHERE Age BETWEEN 20 AND 30;

* Output:

|  |  |
| --- | --- |
| **NAME** | **ADDRESS** |
| SUJIT | Rohtak |
| SUJIT | Rohtak |

**LIKE** operator

It is used to fetch filtered data by searching for a particular pattern in where clause. **Basic Syntax:** **SELECT column1,column2 FROM table\_name WHERE column\_name LIKE pattern;**

**LIKE:** operator name

**pattern:** exact value extracted from the pattern to get related data in result set. **Note**: The character(s) in pattern are case sensitive.

**Queries**

* To fetch records of students where NAME starts with letter S.

SELECT \* FROM Student WHERE NAME LIKE 'S%';

* The ‘%'(wildcard) signifies the later characters here which can be of any length and value.More about wildcards will be discussed in the later set. Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **Age** |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 4 | SURESH | Delhi | XXXXXXXXXX | 18 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |

* To fetch records of students where NAME contains the pattern ‘AM’.

SELECT \* FROM Student WHERE NAME LIKE '%AM%';

* Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **Age** |
| 1 | Ram | Delhi | XXXXXXXXXX | 18 |
| 2 | RAMESH | GURGAON | XXXXXXXXXX | 18 |
| 2 | RAMESH | GURGAON | XXXXXXXXXX | 18 |

**IN** operator

It is used to fetch filtered data same as fetched by ‘=’ operator just the difference is that here we can specify multiple values for which we can get the result set. **Basic Syntax:** **SELECT column1,column2 FROM table\_name WHERE column\_name IN (value1,value2,..);**

**IN:** operator name

**value1,value2,..:** exact value matching the values given and get related data in result set.

**Queries**

* To fetch NAME and ADDRESS of students where Age is 18 or 20.

SELECT NAME,ADDRESS FROM Student WHERE Age IN (18,20);

* Output:

|  |  |
| --- | --- |
| **NAME** | **ADDRESS** |
| Ram | Delhi |
| RAMESH | GURGAON |
| SUJIT | ROHTAK |
| SURESH | Delhi |
| SUJIT | ROHTAK |
| RAMESH | GURGAON |

* To fetch records of students where ROLL\_NO is 1 or 4.

SELECT \* FROM Student WHERE ROLL\_NO IN (1,4);

* Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **Age** |
| 1 | Ram | Delhi | XXXXXXXXXX | 18 |
| 4 | SURESH | Delhi | XXXXXXXXXX | 18 |

**MySQL subquery**

* MySQL subquery is a query nested within another query such as [SELECT](https://www.mysqltutorial.org/mysql-select-statement-query-data.aspx), [INSERT](https://www.mysqltutorial.org/mysql-insert-statement.aspx), [UPDATE](https://www.mysqltutorial.org/mysql-update-data.aspx) or [DELETE](https://www.mysqltutorial.org/mysql-delete-statement.aspx). Also, a subquery can be nested within another subquery. For example, the following query uses a subquery to return the employees who work in the offices located in the USA.
* A MySQL subquery is called an inner query while the query that contains the subquery is called an outer query. A subquery can be used anywhere that expression is used and must be closed in parentheses.

**EXAMPLE**

For example, the following query uses a subquery to return the employees who work in the offices located in the USA.

SELECT   
 lastName, firstName  
FROM  
 employees  
WHERE  
 officeCode IN (SELECT   
 officeCode  
 FROM offices  
 WHERE  
 country = 'USA');

**SQL CONDITIONAL STATEMENTS**

1)**The CASE Expression**

* In a simple CASE expression, the SQL searches for the first WHEN……THEN pair for which expr is equal to comparison\_expr and returns return\_expr.
* If above condition is not satisfied, an ELSE clause exists, the SQL returns else\_expr. Otherwise, returns NULL.

**Syntax: CASE** expr **WHEN** comparison\_expr1 **THEN** return\_expr1  
 [**WHEN** comparison\_expr2 **THEN** return\_expr2  
 .  
 .  
 .  
**WHEN** comparison\_exprn **THEN** return\_exprn  
**ELSE** else\_expr]  
END

**2)GREATEST**

* Returns the largest value from a list of any number of expressions.
* **Syntax: GREATEST(**expr1, expr2 [, .....] **)**
* **Input:**  
  **SELECT GREATEST('XYZ', '**xyz')  
  from dual;
* **Output:**  
  **GREATEST('XYZ', 'xyz')**  
  **xyz**

**3) IN**

Checks whether a value is present within a set of values and can be used with WHERE, CHECK and creation of views.

•**Syntax: WHERE** column **IN (**x1, x2, x3 [,......] **)**

•**Input:**  
SELECT \* from Employee  
 WHERE department\_id IN(50, 120)

# Java JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

* JDBC-ODBC Bridge Driver,
* Native Driver,
* Network Protocol Driver, and
* Thin Driver

We can use JDBC API to access tabular data stored in any relational database.

By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

* Driver interface
* Connection interface
* Statement interface
* PreparedStatement interface
* CallableStatement interface
* ResultSet interface
* ResultSetMetaData interface
* DatabaseMetaData interface
* RowSet interface

### Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

# JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver

2. Native-API driver (partially java driver)

3. Network Protocol driver (fully java driver)

4. Thin driver (fully java driver)

### 1) JDBC-ODBC bridge driver

* The JDBC-ODBC bridge driver uses ODBC driver to connect to the database.
* The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.
* This is now discouraged because of thin driver.

### 2) Native-API driver

* The Native API driver uses the client-side libraries of the database.
* The driver converts JDBC method calls into native calls of the database API.
* It is not written entirely in java.

### 3) Network Protocol driver

* The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol.
* It is fully written in java.

### 4) Thin driver

* The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver.
* It is fully written in Java language.

# Java Database Connectivity with 5 Steps

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

* Register the Driver class
* Create connection
* Create statement
* Execute queries
* Close connection

### 1) Register the driver class

The **forName()** method of Class class is used to register the driver class

This method is used to dynamically load the driver class.

### Syntax of forName() **public** **static** **void** forName(String className)**throws** ClassNotFoundException

### 2 )Create the connection object

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

### Syntax of getConnection() method

### **public** **static** Connection getConnection(String url)**throws** SQLException

### **public** **static** Connection getConnection(String url,String name,String password

### 3) Create the Statement object

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

### Syntax of createStatement() method

### **public** Statement createStatement()**throws** SQLException

### 4) Execute the query

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

### Syntax of executeQuery() method

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

### 5) Close the connection object

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

### Syntax of close() method

**public** **void** close()**throws** SQLException

