UNIT-4

Exception Handlng
Multi threading
Files and I/O streams
Java Database Connection

Exception Handlng:

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

What is Exception?

Exception is an abnormal condition.
In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in java.

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. compile-time (Checked Exception)
2. runtime (Unchecked Exception)
3. Error (logical errors)

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

The **advantages of Exception Handling in Java** are as follows:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

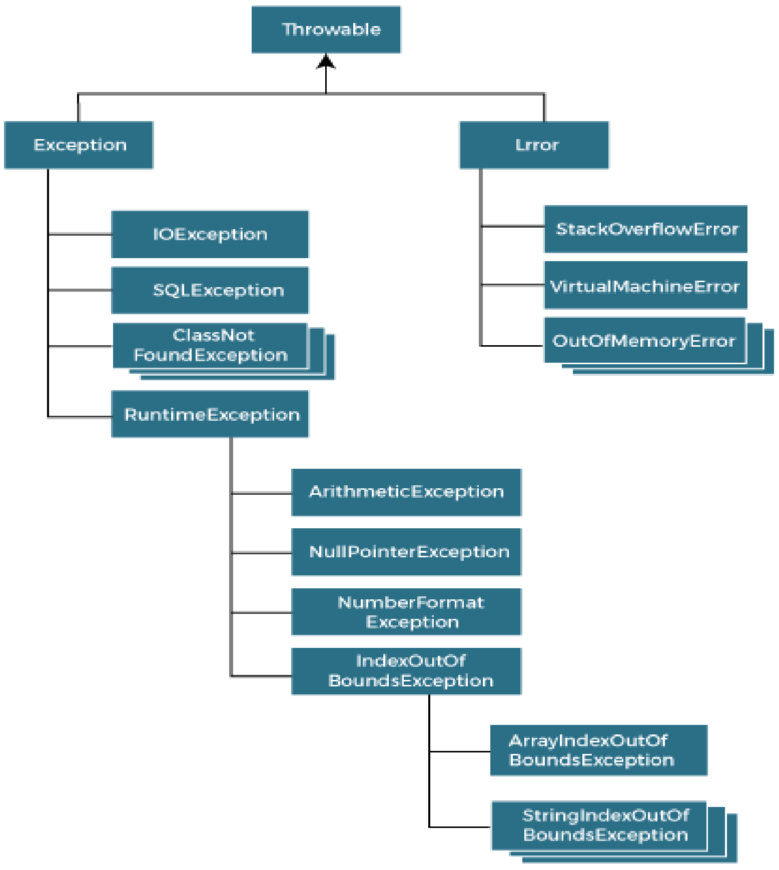
Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```
public class JavaExceptionExample {
public static void main(String args[]) {
try {
 // code that may raise exception
int data = 100 / 0; // This line will throw ArithmeticException
 }
 catch (ArithmeticException e) {
 System.out.println(e); // Print the exception message
} // The rest of the code continues to execute System.out.println("rest of the code...");
} }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

1. Int a=50/0;   //ArithmeticException

2) A scenario where NullPointerException occurs

If we have a null value in any variable performing any operation on the variable throws a NullPointerException.

String s=null;

System.out.println(s.length());   //NullPointerException.

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
String s="abc";

int i=Integer.parseInt(s);    //NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
Int a[]=new int[5];
a[10]=50;    //ArrayIndexOutOfBoundsException
```

try-catch block

try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}
catch(Exception_class_Name ref){
}
```

Syntax of try-finally block

```
try{
//code that may throw an exception
}
finally{\
}
```

catch block

catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

Example

TryCatchExample.java

```
public class TryCatchExample {
 public static void main(String[] args) {
 int data=50/0;          //may throw exception
System.out.println("rest of the code");
 } }
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

Example

TryCatchExample.java

```
public class TryCatchExample2{
public static void main(String[] args) {
try
{
int data=50/0;             //may throw exception
}
//handling the exception
catch(ArithmeticException e)
{
System.out.println(e);
}
System.out.println("rest of the code");
}

}
```

Catch Multiple Exceptions

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

EXAMPLE:

```
public class MultipleCatchBlock {

 public static void main(String[] args) {

try {

int a[] = new int[5];

 a[5] = 30 / 0;              // This line may throw ArithmeticException or
ArrayIndexOutOfBoundsException

 }
 catch (ArithmeticException e) {
 System.out.println("Arithmetic Exception occurs");
 }
 catch (ArrayIndexOutOfBoundsException e{
 System.out.println("ArrayIndexOutOfBounds Exception occurs");
}
 catch (Exception e) {
 System.out.println("Parent Exception occurs");
}
 System.out.println("rest of the code");
} }
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Nested try Syntax:

```
try {

statement 1;

//try catch block within another try block

try {

statement 2;

//try catch block within nested try block

try {
 statement 3;

}

 catch(Exception e1) {

// Handle exception e1(innermost try-catch)

}

 } catch(Exception e2) {

// Handle exception e2 (second level try-catch)

}

 } catch(Exception e3) {

// Handle exception e3 (outermost try-catch)

}
```

Finally block

finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

```
finally {

        scanner.close(); // Close the scanner to prevent resource leak


   }
```

throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

The syntax of the Java throw keyword is given below.

```
Throw new exception_class("error message");
```

the example of throw IOException.

```
Throw new IOException("device error");
```

throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

```
return_type method_name() throws exception_class_name{
        //method code
        }
```

Advantage of Java throws keyword

It provides information to the caller of the method about the exception.

There are many differences between **throw** and **throws** keywords. A list of differences between throw and throws are given below:

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Usage	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.

3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Throw keyword example

Example:

```
//Custom exception class for Insufficient Balance

class InsufficientBalanceException extends Exception {

    public InsufficientBalanceException(String message) {

        super(message);

    }

}

//Account class with withdraw method

class Account {

    private double balance;

    // Constructor

    public Account(double balance) {
        this.balance = balance;

    }

    // Method to withdraw amount

    public void withdraw(double amount) {

    try {

        if (amount > balance) {

            throw new InsufficientBalanceException("Insufficient balance in account");

        }

        // Code to withdraw the amount

        balance -= amount;

        System.out.println("Amount withdrawn: " + amount);

        System.out.println("Remaining balance: " + balance);

        } catch (InsufficientBalanceException e) {

            // Handle InsufficientBalanceException

            System.out.println("Error: " + e.getMessage());
        }

    }

}


public class ThrowDemo {

    public static void main(String[] args) {

        // Create an account with initial balance

        Account account = new Account(1000);

        // Attempt to withdraw an amount

        account.withdraw(1500);

    }

}
```

Throws keyword example:

```
public class ThrowsDemo {

    public static double divide(int numerator, int denominator) throws
    ArithmeticException {

        if (denominator == 0) {
```

```java
            throw new ArithmeticException("Denominator cannot be zero");
        }
        return (double) numerator / denominator;
    }
    public static void main(String[] args) {
        try {
            double result = divide(10, 0);
            System.out.println("Result of division: " + result);
        } catch (ArithmeticException e) {
            System.err.println("Error: " + e.getMessage());
            // Handle the exception, or rethrow it if necessary
        }
    }
}
```

MULTI THREADING:

there are two distinct types of multitasking: process-based and thread-based.

Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.

Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum. This is especially important for the interactive, networked environment in which Java operates because idle time is common

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution.

To create a new thread, your program will either extend Thread or implement the Runnable interface.

The Thread class defines several methods that help manage threads. Some of the commonly used methods are shown here

Java Thread Methods

S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.

5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	suspend()	It is used to suspend the thread.
14)	void	resume()	It is used to resume the suspended thread.

In the most general sense, you create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

public void run( )

Inside run( ), you will define the code that constitutes the new thread. This thread will end when run( ) returns.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. Two of them is shown here:

Thread(Runnable threadOb, String threadName)

Thread(Runnable threadOb)

In these constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) initiates a call to run( ). The start( ) method is shown here:

```
void start( )
```

Example program to create thread with Runnable interface

```
public class MyThread implements Runnable {

        private int i;

        public static void main(String[] args) {

                MyThread t1 = new MyThread();

                MyThread t2 = new MyThread();

                Thread thread1 = new Thread(t1);

                Thread thread2 = new Thread(t2);

                thread1.start();

                thread2.start();

        }
```

```
    @Override

    public void run() {

        for (i = 0; i < 5; i++) {

            try {

                Thread.sleep(500);

                System.out.println(Thread.currentThread().getName() + ": " + i);

            } catch (InterruptedException ex) {

                System.out.println(ex);

            }

        }

    }

}
```

This code defines a class MyThread that implements the Runnable interface. The run() method contains a loop that iterates five times, printing the value of i each time with a 500 milliseconds delay between iterations. If a InterruptedException occurs during the sleep, it's caught and printed.

Thread Priority in Multithreading:

Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by the programmer explicitly.

Priorities in threads is a concept where each thread is having a priority which in layman's language one can say every object is having priority here which is represented by numbers ranging from 1 to 10.

- The default priority is set to 5 as excepted.
- Minimum priority is set to 1.
- Maximum priority is set to 10.

We will use *currentThread()* method to get the name of the current thread. User can also use *setName()* method if he/she wants to make names of thread as per choice for understanding purposes.

- *getName()* method will be used to get the name of the thread.

The accepted value of priority for a thread is in the range of 1 to 10.

public final int getPriority(): java.lang.Thread.getPriority() method returns priority of given thread.

public final void setPriority(int newPriority): java.lang.Thread.setPriority() method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

Extending Thread

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. As before, a call to start( ) begins execution of the new thread. Here is the preceding program rewritten to extend Thread:

```
public class MyThread extends Thread {

    public static void main(String[] args) {


        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();

        t2.start();

    }

    @Override

    public void run() {

        for (int i = 0; i < 5; i++) {

            try {

                Thread.sleep(500);

                System.out.println(Thread.currentThread().getName() + ": " + i);

            } catch (InterruptedException ex) {

                System.out.println(ex);

            }

        }

    }
```

}

Life Cycle of a Thread:

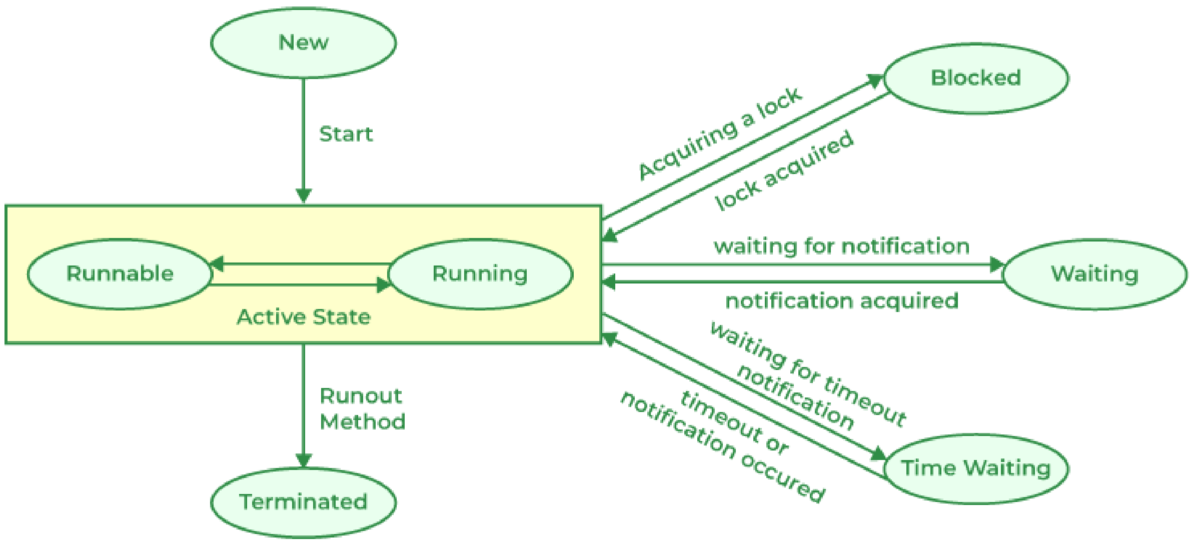
A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

There are multiple states of the thread in a lifecycle as mentioned below:

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.
2. **Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
3. **Blocked:** The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.
4. **Waiting state:** The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.

5. Terminated State: A thread terminates because of either of the following reasons:

- Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
- Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.



Syncronization:

Thread synchronization in Java is essential for managing multiple threads that access shared resources, preventing interference and ensuring consistency.

Why Use Synchronization?

1. To prevent interference between threads.
2. To maintain consistency in data.

Types of Synchronization

1. **Process Synchronization**: Ensures multiple processes coordinate to avoid conflicts and maintain consistency.
2. **Thread Synchronization**: Coordinates the execution of threads to ensure proper resource sharing and prevent data inconsistency in a multithreaded environment.

Thread Synchronization

This concept ensures that **only one thread executes at a time while others wait**, avoiding interference and inconsistency.

Mutual Exclusion

It prevents threads from interfering and maintains distance between them. Achieved using:

- Synchronized Method
- Synchronized Block
- Static Synchronization

Concept of Lock in Java

Synchronization in Java involves a lock or monitor associated with each object. A thread must acquire this lock before accessing an object's fields and release it after finishing. This ensures only one thread can access the synchronized code block at a time, preventing data corruption and inconsistency.

Synchronized Method

When you use a synchronized method, the method itself is synchronized. This means that the whole method is locked for any thread access, ensuring that only one thread can execute this method at a time for a given object instance.

```
class Table{
 synchronized void printTable(int n){//synchronized method
 for(int i=1;i<=5;i++){
  System.out.println(n+"x"+i+"="+n*i);
 try{
 Thread.sleep(400);
  }catch(Exception e){System.out.println(e);}
 }
}
}
class MyThread extends Thread{
Table t;
int v;
MyThread(Table t,int v){
this.t=t;
this.v =v;
}
public void run(){
t.printTable(v);
}
}
public class Synchronization{
public static void main(String args[]){
Table obj = new Table();//only one object(Shared)
MyThread t1=new MyThread(obj, 2);
MyThread t2=new MyThread(obj, 10);
t1.start();
t2.start();
}
}
```

Synchronized Block

A synchronized block is used to lock a specific part of the code rather than the entire method. This provides more granular control over which sections of code need to be synchronized, potentially improving performance by allowing other threads to execute non-critical sections of code concurrently.

```
class Table{

void printTable(int n){
synchronized(this){ //synchronized block
for(int i=1;i<=5;i++){
 System.out.println(n+"x"+i+"="+n*i);
try{
```

```
Thread.sleep(400);
}catch(Exception e){System.out.println(e);}
}
}
}
}
```

Static Synchronization

Static synchronization in Java is used to synchronize static methods. When a static method is synchronized, it locks the class object, meaning that only one thread can execute any synchronized static method in the class at a time. This is useful when you need to control access to static fields or methods of the class, ensuring thread safety at the class level.

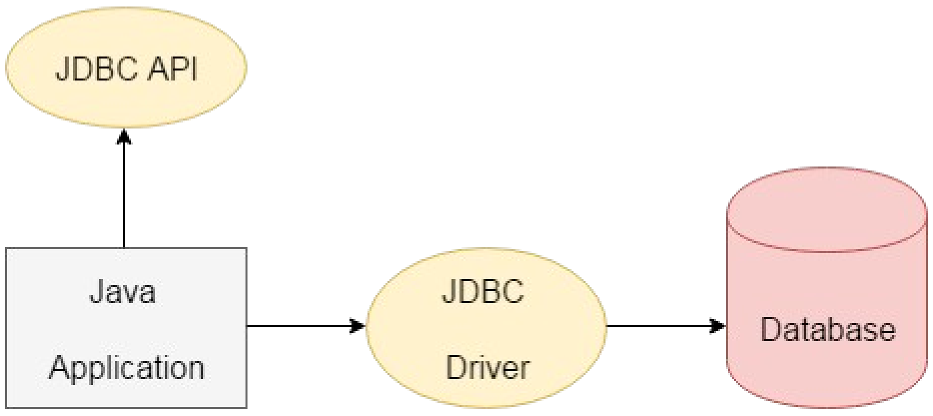
```
class Table1{
static synchronized void printTable(int n){ // synchronized static method
for(int i=1;i<=5;i++){
System.out.println(n+"x"+i+"="+n*i);
try{
Thread.sleep(400);
}catch(Exception e){System.out.println(e);}
}
}
}
```

JDBC:

JDBC stands for **Java Database Connectivity. JDBC** is a **Java API** to connect and execute the query with the database. It is a specification from Sun Microsystems that provides a standard abstraction(API or Protocol) for Java applications to communicate with various databases. It provides the language with Java database connectivity standards. It is used to write programs required to access databases. JDBC, along with the database driver, can access databases and spreadsheets. The enterprise data stored in a relational database(RDB) can be accessed with the help of JDBC APIs.

Definition of JDBC(Java Database Connectivity)

JDBC is an API(Application programming interface) used in Java programming to interact with databases. The **classes** *and* ***interfaces*** of JDBC allow the application to send requests made by users to the specified database. The current version of JDBC is *JDBC 4.3, released on 21st September 2017.*



Purpose of JDBC

Enterprise applications created using the JAVA EE technology need to interact with databases to store application-specific information. So, interacting with a database requires efficient database connectivity, which can be achieved by using the **ODBC**(Open database connectivity) driver. This driver is used with JDBC to interact or communicate with various kinds of databases such as Oracle, MS Access, Mysql, and SQL server database.

Components of JDBC

There are generally four main components of JDBC through which it can interact with a database. They are as mentioned below:

1. JDBC API: It provides various methods and interfaces for easy communication with the database. It provides two packages as follows, which contain the java SE and Java EE platforms to exhibit WORA(write once run anywhere) capabilities.

The **java.sql** package contains interfaces and classes of JDBC API.

java.sql: This package provides APIs for data access and data process in a relational database, included in
Java Standard Edition (java SE)
javax.sql: This package extends the functionality of java package by providing datasource interface for
establishing connection pooling, statement pooling with a data source, included in
Java Enterprise Edition (java EE)

2. JDBC Driver manager: It loads a database-specific driver in an application to establish a connection with a database. It is used to make a database-specific call to the database to process the user request.

3. JDBC Test suite: It is used to test the operation(such as insertion, deletion, updation) being performed by JDBC Drivers.

4. JDBC-ODBC Bridge Drivers: It connects database drivers to the database. This bridge translates the JDBC method call to the ODBC function call. It makes use of the **sun.jdbc.odbc** package which includes a native library to access ODBC characteristics.

What is API?

Before jumping into JDBC Drivers, let us know more about API.

API stands for **Application Programming Interface**. It is essentially a set of rules and protocols which transfers data between different software applications and allow different software applications to communicate with each other. Through an API one application can request information or perform a function from another application without having direct access to it's underlying code or the application data.

JDBC API uses JDBC Drivers to connect with the database.

How to create a Java JDBC connection

We have learned what a JDBC is, what it is used for, its components, driver types, and its relationship with relational databases like SQL. Now, we will be going through detailed steps through which we use JDBC to create a connection to a database in Java. Here is the 7-step process to create a Java JDBC connection:

1. Import the packages:

This includes uploading all the packages containing the JDBC classes, interfaces, and subclasses used during the database programming. More often than not, using import java.sql.* is enough. However, other classes can be imported if needed in the program.

2. Register the drivers:

Before connecting to the database, we'll need to load or register the drivers once per database. This is done to create a communication channel with the database. Loading a driver can be done in two ways:

Class.forName()
DriverManager.registerDriver()

3. Establish a connection:

For the next step here, the getConnection() method is used to create a connection object that will correspond to a physical connection with the database. To get the getConnection() to access the database, the three parameters are a username, string data type URL, and a password. Two methods can be used to achieve this:

- *getConnection(URL, username, password):* This uses three parameters URL, a password, and a username
- *getConnection(URL):* This has only one parameter - URL. The URL has both a username and password. There are several JDBC connection strings for different relational databases and some are listed below:

a. **IBM DB2 database**: jdbc:db2://HOSTNAME:PORT/DATABASE_NAME
b. **Oracle database**: jdbc:oracle:thin:@HOST_NAME:PORT:SERVICE_NAME
c. **My SQL database**: jdbc:mysql://HOST_NAME:PORT/DATABASE_NAME

4. Create a statement:

The statement can now be created to perform the SQL query when the connection has been established. There are three statements from the createStatement method of the connection class to establish the query. These statements are

- *Statement:* This is used to create simple SQL statements with no parameter. An example is: Statement statemnt1 = conn.createStatement();. This statement returns the ResultSet object.
- *PreparedStatement:* This extends the Statement interface. It improves the application's performance because it has more features and compiles the query only once. It is used for precompiled SQL statements that have parameters.
- *CallableStatement:* CallableStatements also extends the PreparedStatement interface. It is used for SQL statements with parameters that invoke procedure or function in the database. It is simply created by calling the prepare all method of the connection object.

5. Execute the query:

This uses a type statement object to build and submit SQL statements to a database. It has four distinct methods:

- ResultSet executeQuery(String sql)
- executeUpdate(String sql)
- execute(String sql)
- executeBatch()

6. Retrieve results or(processing result):

When queries are executed using the executeQuery() method, it produces results stored in the ResultSet object. The ResultSet object is then used to access the retrieved data from the database.

7. Close the connections:

The JDBC connection can now be closed after all is done. The resource has to be closed to avoid running out of connections. It can be done automatically using 'conn.close();'. But for versions of Java 7 and above, it can be closed using a try-catch block.

Example for JDBC:

```
import java.sql.*;        //step-1
public class JDBCconn {
        public static void main(String[] args) {
                String DB_URL = "jdbc:mysql://localhost/TEST";
                String USER = "guest";
                String PASS = "guest123";
                String QUERY = "SELECT * FROM Employee";
```

```
            // Open a connection
            try
            {
                class.forName("com.mysql.jdbc.Driver");         //step-2
                Connection conn = DriverManager.getConnection(DB_URL, USER, PASS); //step-3
                Statement stmt = conn.createStatement();   //step-4
                ResultSet rs = stmt.executeQuery(QUERY);   //step-5
              // Extract data from result set
              while (rs.next()) {   //step-6
                // Retrieve by column name
                System.out.print(rs.getInt(1) +" "+rs.getString(2)+" "+rs.getString(3));
                 conn.close();    //step-7

              }
            } catch (SQLException e) {
              System.out.println(e);
            }

          }
}
```
