Sirf khwabh Nhi

31/08

1) Java Exceptions

we will learn about exceptions in Java. We will cover errors, exceptions and different types of exceptions in Java.

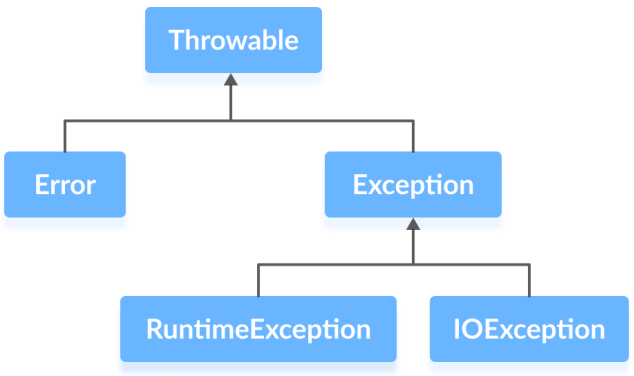
An exception is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Java Exception hierarchy

Here is a simplified diagram of the exception hierarchy in Java.



As you can see from the image above, the `Throwable` class is the root class in the hierarchy.

Note that the hierarchy splits into two branches: Error and Exception.

Errors

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

Errors are usually beyond the control of the programmer and we should not try to handle errors.

Exceptions

Exceptions can be caught and handled by the program.

When an exception occurs within a method, it creates an object. This object is called the exception object.

It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred.

We will learn how to handle these exceptions in the next tutorial. In this tutorial, we will now focus on different types of exceptions in Java.

Java Exception Types

The exception hierarchy also has two branches: `RuntimeException` and `IOException`.

---

1. RuntimeException

A runtime exception happens due to a programming error. They are also known as unchecked exceptions.

These exceptions are not checked at compile-time but run-time. Some of the common runtime exceptions are:

- Improper use of an API - `IllegalArgumentException`
- Null pointer access (missing the initialization of a variable) - `NullPointerException`
- Out-of-bounds array access - `ArrayIndexOutOfBoundsException`
- Dividing a number by 0 - `ArithmeticException`

You can think about it in this way. “If it is a runtime exception, it is your fault”.

The `NullPointerException` would not have occurred if you had checked whether the variable was initialized or not before using it.

An `ArrayIndexOutOfBoundsException` would not have occurred if you tested the array index against the array bounds.

---

2. IOException

An `IOException` is also known as a checked exception. They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions.

Some of the examples of checked exceptions are:

- Trying to open a file that doesn’t exist results in `FileNotFoundException`
- Trying to read past the end of a file

2)Java Exception Handling

we will learn about different approaches of exception handling in Java with the help of examples.

We know that exceptions abnormally terminate the execution of a program.

This is why it is important to handle exceptions. Here's a list of different approaches to handle exceptions in Java.

- try...catch block
- finally block
- throw and throws keyword

---

1. Java try...catch block

The try-catch block is used to handle exceptions in Java. Here's the syntax of `try...catch` block:

```
try {

  // code

}

catch(Exception e) {

  // code

}
```

Here, we have placed the code that might generate an exception inside the `try` block. Every `try` block is followed by a `catch` block.

When an exception occurs, it is caught by the `catch` block. The `catch` block cannot be used without the `try` block.

Example: Exception handling using try...catch

```
class Main {

  public static void main(String[] args) {
```

```
    try {

        // code that generate exception

        int divideByZero = 5 / 0;

        System.out.println("Rest of code in try block");

    }


    catch (ArithmeticException e) {

        System.out.println("ArithmeticException => " + e.getMessage());

    }

}
```

```
}
```

Output

```
ArithmeticException => / by zero
```

In the example, we are trying to divide a number by `0`. Here, this code generates an exception.

To handle the exception, we have put the code, `5 / 0` inside the `try` block. Now when an exception occurs, the rest of the code inside the `try` block is skipped.

The `catch` block catches the exception and statements inside the catch block is executed.

If none of the statements in the `try` block generates an exception, the `catch` block is skipped.

---

2. Java finally block

In Java, the `finally` block is always executed no matter whether there is an exception or not.

The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.

The basic syntax of `finally` block is:

```
try {


  //code


}


catch (ExceptionType1 e1) {


  // catch block
```

```
}

finally {

  // finally block always executes

}
```

If an exception occurs, the `finally` block is executed after the `try...catch` block. Otherwise, it is executed after the try block. For each `try` block, there can be only one `finally` block.

Example: Java Exception Handling using finally block

```
class Main {

  public static void main(String[] args) {

    try {

      // code that generates exception

      int divideByZero = 5 / 0;

    }
```

```
    catch (ArithmeticException e) {

        System.out.println("ArithmeticException => " + e.getMessage());

    }

    finally {

        System.out.println("This is the finally block");

    }

  }

}
```

Output

```
ArithmeticException => / by zero
```

```
This is the finally block
```

In the above example, we are dividing a number by 0 inside the `try` block. Here, this code generates an `ArithmeticException`.

The exception is caught by the `catch` block. And, then the `finally` block is executed.

Note: It is a good practice to use the `finally` block. It is because it can include important cleanup codes like,

- code that might be accidentally skipped by return, continue or break
- closing a file or connection

---

3. Java throw and throws keyword

The Java `throw` keyword is used to explicitly throw a single exception.

When we `throw` an exception, the flow of the program moves from the `try` block to the `catch` block.

Example: Exception handling using Java throw

```
class Main {



  public static void divideByZero() {









    // throw an exception



    throw new ArithmeticException("Trying to divide by 0");
```

```
  }


  public static void main(String[] args) {


    divideByZero();


  }


}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by
0


        at Main.divideByZero(Main.java:5)


        at Main.main(Main.java:9)
```

In the above example, we are explicitly throwing the `ArithmeticException` using the `throw` keyword.

Similarly, the `throws` keyword is used to declare the type of exceptions that might occur within the method. It is used in the method declaration.

Example: Java throws keyword

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

```
class Main {

  // declareing the type of exception

  public static void findFile() throws IOException {




    // code that may generate IOException

    File newFile = new File("test.txt");

    FileInputStream stream = new FileInputStream(newFile);

  }




  public static void main(String[] args) {
```

```
    try {

        findFile();

    }

    catch (IOException e) {

        System.out.println(e);

    }

  }

}
```

Output

```
java.io.FileNotFoundException: test.txt (The system cannot find the file specified)
```

When we run this program, if the file test.txt does not exist, `FileInputStream` throws a `FileNotFoundException` which extends the `IOException` class.

The `findFile()` method specifies that an `IOException` can be thrown. The `main()` method calls this method and handles the exception if it is thrown.

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the `throws` clause.

3)Java try...catch

we will learn about the try catch statement in Java with the help of examples.

The `try...catch` block in Java is used to handle exceptions and prevents the abnormal termination of the program.

Here's the syntax of a `try...catch` block in Java.

```
try{
  // code
}
catch(exception) {
  // code
}
```

The `try` block includes the code that might generate an exception.

The `catch` block includes the code that is executed when there occurs an exception inside the `try` block.

Example: Java try...catch block

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


    try {
      int divideByZero = 5 / 0;
      System.out.println("Rest of code in try block");
    }
```

```
    catch (ArithmeticException e) {

      System.out.println("ArithmeticException => " + e.getMessage());

    }

  }

}
```

Output

```
ArithmeticException => / by zero
```

In the above example, notice the line,

```
int divideByZero = 5 / 0;
```

Here, we are trying to divide a number by zero. In this case, an exception occurs. Hence, we have enclosed this code inside the `try` block.

When the program encounters this code, `ArithmeticException` occurs. And, the exception is caught by the `catch` block and executes the code inside the `catch` block.

The `catch` block is only executed if there exists an exception inside the `try` block.

> Note: In Java, we can use a `try` block without a `catch` block. However, we cannot use a `catch` block without a `try` block.

Java try...finally block

We can also use the `try` block along with a finally block.

In this case, the finally block is always executed whether there is an exception inside the try block or not.

Example: Java try...finally block

```
class Main {

  public static void main(String[] args) {
```

```
    try {
      int divideByZero = 5 / 0;
    }

    finally {
      System.out.println("Finally block is always executed");
    }
  }
}
```

Output

```
Finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:4)
```

In the above example, we have used the `try` block along with the `finally` block. We can see that the code inside the `try` block is causing an exception.

However, the code inside the `finally` block is executed irrespective of the exception.

Java try...catch...finally block

In Java, we can also use the finally block after the `try...catch` block. For example,

```
import java.io.*;

class ListOfNumbers {

  // create an integer array
```

```
  private int[] list = {5, 6, 8, 9, 2};

  // method to write data from array to a fila
  public void writeList() {
    PrintWriter out = null;

    try {
      System.out.println("Entering try statement");

      // creating a new file OutputFile.txt
      out = new PrintWriter(new FileWriter("OutputFile.txt"));

      // writing values from list array to Output.txt
      for (int i = 0; i < 7; i++) {
        out.println("Value at: " + i + " = " + list[i]);
      }
    }

    catch (Exception e) {
      System.out.println("Exception => " + e.getMessage());
    }

    finally {
```

```
      // checking if PrintWriter has been opened
      if (out != null) {
        System.out.println("Closing PrintWriter");
        // close PrintWriter
        out.close();
      }

      else {
        System.out.println("PrintWriter not open");
      }
    }

  }
}

class Main {
  public static void main(String[] args) {
    ListOfNumbers list = new ListOfNumbers();
    list.writeList();
  }
}
```

Output

```
Entering try statement
```

```
Exception => Index 5 out of bounds for length 5

Closing PrintWriter
```

In the above example, we have created an array named `list` and a file named `output.txt`. Here, we are trying to read data from the array and storing to the file.

Notice the code,

```
for (int i = 0; i < 7; i++) {

  out.println("Value at: " + i + " = " + list[i]);

}
```

Here, the size of the array is `5` and the last element of the array is at `list[4]`. However, we are trying to access elements at *`a[5]`* and `a[6]`.

Hence, the code generates an exception that is caught by the catch block.

Since the `finally` block is always executed, we have included code to close the `PrintWriter` inside the finally block.

It is a good practice to use finally block to include important cleanup code like closing a file or connection.

Note: There are some cases when a `finally` block does not execute:

- Use of `System.exit()` method
- An exception occurs in the `finally` block
- The death of a thread

Multiple Catch blocks

For each `try` block, there can be zero or more `catch` blocks. Multiple `catch` blocks allow us to handle each exception differently.

The argument type of each `catch` block indicates the type of exception that can be handled by it. For example,

```
class ListOfNumbers {

  public int[] arr = new int[10];



  public void writeList() {



    try {

      arr[10] = 11;

    }



    catch (NumberFormatException e1) {

      System.out.println("NumberFormatException => " + e1.getMessage());

    }



    catch (IndexOutOfBoundsException e2) {

      System.out.println("IndexOutOfBoundsException => " + e2.getMessage());

    }



  }

}



class Main {

  public static void main(String[] args) {

    ListOfNumbers list = new ListOfNumbers();
```

```
    list.writeList();

  }

}
```

Output

```
IndexOutOfBoundsException => Index 10 out of bounds for length 10
```

In this example, we have created an integer array named `arr` of size 10.

Since the array index starts from 0, the last element of the array is at `arr[9]`. Notice the statement,

```
arr[10] = 11;
```

Here, we are trying to assign a value to the index 10. Hence, `IndexOutOfBoundException` occurs.

When an exception occurs in the `try` block,

- The exception is thrown to the first `catch` block. The first `catch` block does not handle an `IndexOutOfBoundsException`, so it is passed to the next `catch` block.
- The second `catch` block in the above example is the appropriate exception handler because it handles an `IndexOutOfBoundsException`. Hence, it is executed.

Catching Multiple Exceptions

From Java SE 7 and later, we can now catch more than one type of exception with one `catch` block.

This reduces code duplication and increases code simplicity and efficiency.

Each exception type that can be handled by the `catch` block is separated using a vertical bar `|`.

Its syntax is:

```
try {

  // code

} catch (ExceptionType1 | Exceptiontype2 ex) {
```

```
  // catch block

}
```

Java try-with-resources statement

The try-with-resources statement is a try statement that has one or more resource declarations.

Its syntax is:

```
try (resource declaration) {

  // use of the resource

} catch (ExceptionType e1) {

  // catch block

}
```

The resource is an object to be closed at the end of the program. It must be declared and initialized in the try statement.

Let's take an example.

```
try (PrintWriter out = new PrintWriter(new FileWriter("OutputFile.txt")) {

  // use of the resource

}
```

The try-with-resources statement is also referred to as automatic resource management. This statement automatically closes all the resources at the end of the statement.

4)Java throw and throws

we will learn to use throw and throws keyword for exception handling with the help of examples.

In Java, exceptions can be categorized into two types:

- Unchecked Exceptions: They are not checked at compile-time but at run-time.For example: `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`, exceptions under `Error` class, etc.
- Checked Exceptions: They are checked at compile-time. For example, `IOException`, `InterruptedException`, etc.

In detail about checked and unchecked exceptions.

Usually, we don't need to handle unchecked exceptions. It's because unchecked exceptions occur due to programming errors. And, it is a good practice to correct them instead of handling them.

This tutorial will now focus on how to handle checked exceptions using `throw` and `throws`.

Java throws keyword

We use the `throws` keyword in the method declaration to declare the type of exceptions that might occur within it.

Its syntax is:

```
accessModifier returnType methodName() throws ExceptionType1, ExceptionType2 ... {

  // code

}
```

As you can see from the above syntax, we can use `throws` to declare multiple exceptions.

Example 1: Java throws Keyword

```
import java.io.*;

class Main {

  public static void findFile() throws IOException {

    // code that may produce IOException

    File newFile=new File("test.txt");

    FileInputStream stream=new FileInputStream(newFile);

  }



  public static void main(String[] args) {

    try{

      findFile();

    } catch(IOException e){

      System.out.println(e);

    }

  }

}
```

Output

```
java.io.FileNotFoundException: test.txt (No such file or directory)
```

When we run this program, if the file `test.txt` does not exist, `FileInputStream` throws a `FileNotFoundException` which extends the `IOException` class.

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the `throws` clause so that methods further up in the call stack can handle them or specify them using `throws` keyword themselves.

The `findFile()` method specifies that an `IOException` can be thrown. The `main()` method calls this method and handles the exception if it is thrown.

Throwing multiple exceptions

Here's how we can throw multiple exceptions using the `throws` keyword.

```
import java.io.*;

class Main {

  public static void findFile() throws NullPointerException, IOException,
InvalidClassException {



    // code that may produce NullPointerException

    … … …



    // code that may produce IOException

    … … …



    // code that may produce InvalidClassException

    … … …

  }



  public static void main(String[] args) {

    try{
```

```
        findFile();
    } catch(IOException e1){
        System.out.println(e1.getMessage());
    } catch(InvalidClassException e2){
        System.out.println(e2.getMessage());
    }
  }
}
```

Here, the `findFile()` method specifies that it can throw `NullPointerException`, `IOException`, and `InvalidClassException` in its `throws` clause.

Note that we have not handled the `NullPointerException`. This is because it is an unchecked exception. It is not necessary to specify it in the `throws` clause and handle it.

---

throws keyword Vs. try...catch...finally

There might be several methods that can cause exceptions. Writing `try...catch` for each method will be tedious and code becomes long and less-readable.

`throws` is also useful when you have checked exception (an exception that must be handled) that you don't want to catch in your current method.

Java throw keyword

The `throw` keyword is used to explicitly throw a single exception.

When an exception is thrown, the flow of program execution transfers from the `try` block to the `catch` block. We use the `throw` keyword within a method.

Its syntax is:

```
throw throwableObject;
```

A throwable object is an instance of class `Throwable` or subclass of the `Throwable` class.

Example 2: Java throw keyword

```
class Main {

  public static void divideByZero() {

    throw new ArithmeticException("Trying to divide by 0");

  }



  public static void main(String[] args) {

    divideByZero();

  }

}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by
0

    at Main.divideByZero(Main.java:3)

    at Main.main(Main.java:7)

exit status 1
```

In this example, we are explicitly throwing an `ArithmeticException`.

Note: `ArithmeticException` is an unchecked exception. It's usually not necessary to handle unchecked exceptions.

Example 3: Throwing checked exception

```
import java.io.*;

class Main {

  public static void findFile() throws IOException {

    throw new IOException("File not found");

  }



  public static void main(String[] args) {

    try {

      findFile();

      System.out.println("Rest of code in try block");

    } catch (IOException e) {

      System.out.println(e.getMessage());

    }

  }

}
```

Output

```
File not found
```

The `findFile()` method throws an `IOException` with the message we passed to its constructor.

Note that since it is a checked exception, we must specify it in the `throws` clause.

The methods that call this `findFile()` method need to either handle this exception or specify it using `throws` keyword themselves.

We have handled this exception in the `main()` method. The flow of program execution transfers from the `try` block to `catch` block when an exception is thrown. So, the rest of the code in the `try` block is skipped and statements in the `catch` block are executed.

5)Java catch Multiple Exceptions

we will learn to handle multiple exceptions in Java with the help of examples.

Before Java 7, we had to write multiple exception handling codes for different types of exceptions even if there was code redundancy.

Let's take an example.

Example 1: Multiple catch blocks

```
class Main {

  public static void main(String[] args) {

    try {

      int array[] = new int[10];

      array[10] = 30 / 0;

    } catch (ArithmeticException e) {

      System.out.println(e.getMessage());

    } catch (ArrayIndexOutOfBoundsException e) {
```

```
        System.out.println(e.getMessage());
    }
  }
}
```

Output

```
/ by zero
```

In this example, two exceptions may occur:

- `ArithmeticException` because we are trying to divide a number by 0.
- `ArrayIndexOutOfBoundsException` because we have declared a new integer array with array bounds 0 to 9 and we are trying to assign a value to index 10.

We are printing out the exception message in both the `catch` blocks i.e. duplicate code.

The associativity of the assignment operator = is right to left, so an `ArithmeticException` is thrown first with the message `/ by zero`.

Handle Multiple Exceptions in a catch Block

In Java SE 7 and later, we can now catch more than one type of exception in a single `catch` block.

Each exception type that can be handled by the `catch` block is separated using a vertical bar or pipe `|`.

Its syntax is:

```
try {
  // code
```

```
} catch (ExceptionType1 | Exceptiontype2 ex) {

  // catch block

}
```

Example 2: Multi-catch block

```
class Main {

  public static void main(String[] args) {

    try {

      int array[] = new int[10];

      array[10] = 30 / 0;

    } catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {

      System.out.println(e.getMessage());

    }

  }

}
```

Output

```
/ by zero
```

Catching multiple exceptions in a single `catch` block reduces code duplication and increases efficiency.

The bytecode generated while compiling this program will be smaller than the program having multiple `catch` blocks as there is no code redundancy.

Note: If a `catch` block handles multiple exceptions, the catch parameter is implicitly `final`. This means we cannot assign any values to catch parameters.

Catching base Exception

When catching multiple exceptions in a single `catch` block, the rule is generalized to specialized.

This means that if there is a hierarchy of exceptions in the `catch` block, we can catch the base exception only instead of catching multiple specialized exceptions.

Let's take an example.

Example 3: Catching base exception class only

```
class Main {

  public static void main(String[] args) {

    try {

      int array[] = new int[10];

      array[10] = 30 / 0;

    } catch (Exception e) {

      System.out.println(e.getMessage());

    }

  }

}
```

Output

```
/ by zero
```

We know that all the exception classes are subclasses of the `Exception` class. So, instead of catching multiple specialized exceptions, we can simply catch the `Exception` class.

If the base exception class has already been specified in the `catch` block, do not use child exception classes in the same `catch` block. Otherwise, we will get a compilation error.

Let's take an example.

Example 4: Catching base and child exception classes

```
class Main {

  public static void main(String[] args) {

    try {

      int array[] = new int[10];

      array[10] = 30 / 0;

    } catch (Exception | ArithmeticException | ArrayIndexOutOfBoundsException
e) {

      System.out.println(e.getMessage());

    }

  }

}
```

Output

```
Main.java:6: error: Alternatives in a multi-catch statement cannot be related
by subclassing
```

In this example, `ArithmeticException` and `ArrayIndexOutOfBoundsException` are both subclasses of the `Exception` class. So, we get a compilation error.

Hee evdha important nhi tari baghun thev

6)Java try-with-resources

we will learn about the try-with-resources statement to close resources automatically.

The `try-with-resources` statement automatically closes all the resources at the end of the statement. A resource is an object to be closed at the end of the program.

Its syntax is:

```
try (resource declaration) {
  // use of the resource
} catch (ExceptionType e1) {
  // catch block
}
```

As seen from the above syntax, we declare the `try-with-resources` statement by,

1. declaring and instantiating the resource within the `try` clause.
2. specifying and handling all exceptions that might be thrown while closing the resource.

Note: The try-with-resources statement closes all the resources that implement the AutoCloseable interface.

Let us take an example that implements the `try-with-resources` statement.

Example 1: try-with-resources

```
import java.io.*;


class Main {
```

```
  public static void main(String[] args) {

    String line;

    try(BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {

      while ((line = br.readLine()) != null) {

        System.out.println("Line =>"+line);

      }

    } catch (IOException e) {

      System.out.println("IOException in try block =>" + e.getMessage());

    }

  }

}
```

Output if the test.txt file is not found.

```
IOException in try-with-resources block =>test.txt (No such file or
directory)
```

Output if the test.txt file is found.

```
Entering try-with-resources block

Line =>test line
```

In this example, we use an instance of `BufferedReader` to read data from the `test.txt` file.

Declaring and instantiating the `BufferedReader` inside the `try-with-resources` statement ensures that its instance is closed regardless of whether the `try` statement completes normally or throws an exception.

If an exception occurs, it can be handled using the exception handling blocks or the throws keyword.

Suppressed Exceptions

In the above example, exceptions can be thrown from the `try-with-resources` statement when:

- The file `test.txt` is not found.
- Closing the `BufferedReader` object.

An exception can also be thrown from the `try` block as a file read can fail for many reasons at any time.

If exceptions are thrown from both the `try` block and the `try-with-resources` statement, exception from the `try` block is thrown and exception from the `try-with-resources` statement is suppressed.

Retrieving Suppressed Exceptions

In Java 7 and later, the suppressed exceptions can be retrieved by calling the `Throwable.getSuppressed()` method from the exception thrown by the `try` block.

This method returns an array of all suppressed exceptions. We get the suppressed exceptions in the `catch` block.

```
catch(IOException e) {

  System.out.println("Thrown exception=>" + e.getMessage());

  Throwable[] suppressedExceptions = e.getSuppressed();

  for (int i=0; i<suppressedExceptions.length; i++) {

    System.out.println("Suppressed exception=>" + suppressedExceptions[i]);

  }
```

```
}
```

Advantages of using try-with-resources

Here are the advantages of using try-with-resources:

1. finally block not required to close the resource

Before Java 7 introduced this feature, we had to use the `finally` block to ensure that the resource is closed to avoid resource leaks.

Here's a program that is similar to Example 1. However, in this program, we have used finally block to close resources.

Example 2: Close resource using finally block

```
import java.io.*;



class Main {

  public static void main(String[] args) {

    BufferedReader br = null;

    String line;



    try {

      System.out.println("Entering try block");

      br = new BufferedReader(new FileReader("test.txt"));

      while ((line = br.readLine()) != null) {

        System.out.println("Line =>"+line);

      }
```

```
    } catch (IOException e) {
      System.out.println("IOException in try block =>" + e.getMessage());
    } finally {
      System.out.println("Entering finally block");
      try {
        if (br != null) {
          br.close();
        }
      } catch (IOException e) {
        System.out.println("IOException in finally block =>"+e.getMessage());
      }

    }
  }
}
```

Output

```
Entering try block
Line =>line from test.txt file
Entering finally block
```

As we can see from the above example, the use of `finally` block to clean up resources makes the code more complex.

Notice the `try...catch` block in the `finally` block as well? This is because an `IOException` can also occur while closing the `BufferedReader` instance inside this `finally` block so it is also caught and handled.

The `try-with-resources` statement does automatic resource management. We need not explicitly close the resources as JVM automatically closes them. This makes the code more readable and easier to write.

---

2. try-with-resources with multiple resources

We can declare more than one resource in the `try-with-resources` statement by separating them with a semicolon `;`

Example 3: try with multiple resources

```
import java.io.*;

import java.util.*;

class Main {

  public static void main(String[] args) throws IOException{

    try (Scanner scanner = new Scanner(new File("testRead.txt"));

      PrintWriter writer = new PrintWriter(new File("testWrite.txt"))) {

      while (scanner.hasNext()) {

        writer.print(scanner.nextLine());

      }

    }

  }

}
```

If this program executes without generating any exceptions, `Scanner` object reads a line from the `testRead.txt` file and writes it in a new `testWrite.txt` file.

When multiple declarations are made, the `try-with-resources` statement closes these resources in reverse order. In this example, the `PrintWriter` object is closed first and then the `Scanner` object is closed.

Java 9 try-with-resources enhancement

In Java 7, there is a restriction to the `try-with-resources` statement. The resource needs to be declared locally within its block.

```
try (Scanner scanner = new Scanner(new File("testRead.txt"))) {

  // code

}
```

If we declared the resource outside the block in Java 7, it would have generated an error message.

```
Scanner scanner = new Scanner(new File("testRead.txt"));

try (scanner) {

  // code

}
```

To deal with this error, Java 9 improved the `try-with-resources` statement so that the reference of the resource can be used even if it is not declared locally. The above code will now execute without any compilation error.

7)Java Annotations

we will learn what annotations are, different Java annotations and how to use them with the help of examples.

Java annotations are metadata (data about data) for our program source code.

They provide additional information about the program to the compiler but are not part of the program itself. These annotations do not affect the execution of the compiled program.

Annotations start with `@`. Its syntax is:

```
@AnnotationName
```

Let's take an example of `@Override` annotation.

The `@Override` annotation specifies that the method that has been marked with this annotation overrides the method of the superclass with the same method name, return type, and parameter list.

It is not mandatory to use `@Override` when overriding a method. However, if we use it, the compiler gives an error if something is wrong (such as wrong parameter type) while overriding the method.

Example 1: @Override Annotation Example

```
class Animal {

  public void displayInfo() {

    System.out.println("I am an animal.");

  }

}


class Dog extends Animal {

  @Override
```

```
  public void displayInfo() {
    System.out.println("I am a dog.");
  }
}

class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.displayInfo();
  }
}
```

Output

```
I am a dog.
```

In this example, the method `displayInfo()` is present in both the superclass `Animal` and subclass `Dog`. When this method is called, the method of the subclass is called instead of the method in the superclass.

Annotation formats

Annotations may also include elements (members/attributes/parameters).

1. Marker Annotations

Marker annotations do not contain members/elements. It is only used for marking a declaration.

Its syntax is:

```
@AnnotationName()
```

Since these annotations do not contain elements, parentheses can be excluded. For example,

```
@Override
```

---

2. Single element Annotations

A single element annotation contains only one element.

Its syntax is:

```
@AnnotationName(elementName = "elementValue")
```

If there is only one element, it is a convention to name that element as `value`.

```
@AnnotationName(value = "elementValue")
```

In this case, the element name can be excluded as well. The element name will be `value` by default.

```
@AnnotationName("elementValue")
```

---

3. Multiple element Annotations

These annotations contain multiple elements separated by commas.

Its syntax is:

```
@AnnotationName(element1 = "value1", element2 = "value2")
```

Annotation placement

Any declaration can be marked with annotation by placing it above that declaration. As of Java 8, annotations can also be placed before a type.

1. Above declarations

As mentioned above, Java annotations can be placed above class, method, interface, field, and other program element declarations.

Example 2: @SuppressWarnings Annotation Example

```
import java.util.*;



class Main {

  @SuppressWarnings("unchecked")

  static void wordsList() {

    ArrayList wordList = new ArrayList<>();



// This causes an unchecked warning

    wordList.add("programiz");



    System.out.println("Word list => " + wordList);

  }



  public static void main(String args[]) {

    wordsList();

  }

}
```

Output

```
Word list => [programiz]
```

If the above program is compiled without using the `@SuppressWarnings("unchecked")` annotation, the compiler will still compile the program but it will give warnings like:

```
Main.java uses unchecked or unsafe operations.

Word list => [programiz]
```

We are getting the warning

```
Main.java uses unchecked or unsafe operations
```

because of the following statement.

```
ArrayList wordList = new ArrayList<>();
```

This is because we haven't defined the generic type of the array list. We can fix this warning by specifying generics inside angle brackets <>.

```
ArrayList<String> wordList = new ArrayList<>();
```

---

2. Type annotations

Before Java 8, annotations could be applied to declarations only. Now, type annotations can be used as well. This means that we can place annotations wherever we use a type.

Constructor invocations

```
new @Readonly ArrayList<>()
```

Type definitions

```
@NonNull String str;
```

This declaration specifies non-null variable `str` of type `String` to avoid `NullPointerException`.

```
@NonNull List<String> newList;
```

This declaration specifies a non-null list of type `String`.

```
List<@NonNull String> newList;
```

This declaration specifies a list of non-null values of type `String`.

Type casts

```
newStr = (@NonNull String) str;
```

extends and implements clause

```
class Warning extends @Localized Message
```

throws clause

```
public String readMethod() throws @Localized IOException
```

Type annotations enable Java code to be analyzed better and provide even stronger type checks.

Types of Annotations

1. Predefined annotations

    1. `@Deprecated`

2. `@Override`
3. `@SuppressWarnings`
4. `@SafeVarargs`
5. `@FunctionalInterface`

2. Meta-annotations

1. `@Retention`
2. `@Documented`
3. `@Target`
4. `@Inherited`
5. `@Repeatable`

3. Custom annotations

Use of Annotations

- Compiler instructions - Annotations can be used for giving instructions to the compiler, detect errors or suppress warnings. The built-in annotations `@Deprecated`, `@Override`,`@SuppressWarnings` are used for these purposes.
- Compile-time instructions - Compile-time instructions provided by these annotations help the software build tools to generate code, XML files and many more.
- Runtime instructions - Some annotations can be defined to give instructions to the program at runtime. These annotations are accessed using Java Reflection.

8)Java Annotation Types

we will learn about different types of Java annotation with the help of examples.

Java annotations are metadata (data about data) for our program source code. There are several predefined annotations provided by the Java SE. Moreover, we can also create custom annotations as per our needs.

These annotations can be categorized as:

1. Predefined annotations

- `@Deprecated`
- `@Override`
- `@SuppressWarnings`
- `@SafeVarargs`
- `@FunctionalInterface`

2. Custom annotations

3. Meta-annotations

- `@Retention`
- `@Documented`
- `@Target`
- `@Inherited`
- `@Repeatable`

Predefined Annotation Types

1. @Deprecated

The`@Deprecated` annotation is a marker annotation that indicates the element (class, method, field, etc) is deprecated and has been replaced by a newer element.

Its syntax is:

```
@Deprecated
accessModifier returnType deprecatedMethodName() { ... }
```

When a program uses the element that has been declared deprecated, the compiler generates a warning.

We use Javadoc `@deprecated` tag for documenting the deprecated element.

```
/**
 * @deprecated
 * why it was deprecated
 */
@Deprecated
accessModifier returnType deprecatedMethodName() { ... }
```

Example 1: @Deprecated annotation example

```
class Main {
  /**
   * @deprecated
   * This method is deprecated and has been replaced by newMethod()
   */
  @Deprecated
  public static void deprecatedMethod() {
    System.out.println("Deprecated method");
```

```
    }


    public static void main(String args[]) {
        deprecatedMethod();
    }
}
```

Output

```
Deprecated method
```

---

2. @Override

The `@Override` annotation specifies that a method of a subclass overrides the method of the superclass with the same method name, return type, and parameter list.

It is not mandatory to use `@Override` when overriding a method. However, if we use it, the compiler gives an error if something is wrong (such as wrong parameter type) while overriding the method.

Example 2: @Override annotation example

```
class Animal {


    // overridden method
    public void display(){
        System.out.println("I am an animal");
    }
}
```

```
class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){
    display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

Output

```
I am a dog
```

In this example, by making an object `dog1` of `Dog` class, we can call its method `printMessage()` which then executes the `display()` statement.

Since `display()` is defined in both the classes, the method of subclass `Dog` overrides the method of superclass `Animal`. Hence, the `display()` of the subclass is called.

---

3. @SuppressWarnings

As the name suggests, the `@SuppressWarnings` annotation instructs the compiler to suppress warnings that are generated while the program executes.

We can specify the type of warnings to be suppressed. The warnings that can be suppressed are compiler-specific but there are two categories of warnings: deprecation and unchecked.

To suppress a particular category of warning, we use:

```
@SuppressWarnings("warningCategory")
```

For example,

```
@SuppressWarnings("deprecated")
```

To suppress multiple categories of warnings, we use:

```
@SuppressWarnings({"warningCategory1", "warningCategory2"})
```

For example,

```
@SuppressWarnings({"deprecated", "unchecked"})
```

Category `deprecated` instructs the compiler to suppress warnings when we use a deprecated element.

Category `unchecked` instructs the compiler to suppress warnings when we use raw types.

And, undefined warnings are ignored. For example,

```
@SuppressWarnings("someundefinedwarning")
```

Example 3: @SuppressWarnings annotation example

```
class Main {

  @Deprecated

  public static void deprecatedMethod() {

    System.out.println("Deprecated method");

  }

  @SuppressWarnings("deprecated")

  public static void main(String args[]) {

    Main depObj = new Main();

    depObj. deprecatedMethod();

  }

}
```

Output

```
Deprecated method
```

Here, `deprecatedMethod()` has been marked as deprecated and will give compiler warnings when used. By using the `@SuppressWarnings("deprecated")` annotation, we can avoid compiler warnings.

---

4. @SafeVarargs

The `@SafeVarargs` annotation asserts that the annotated method or constructor does not perform unsafe operations on its varargs (variable number of arguments).

We can only use this annotation on methods or constructors that cannot be overridden. This is because the methods that override them might perform unsafe operations.

Before Java 9, we could use this annotation only on final or static methods because they cannot be overridden. We can now use this annotation for private methods as well.

Example 4: @SafeVarargs annotation example

```
import java.util.*;


class Main {


  private void displayList(List<String>... lists) {

    for (List<String> list : lists) {

      System.out.println(list);

    }

  }


  public static void main(String args[]) {

    Main obj = new Main();


    List<String> universityList = Arrays.asList("Tribhuvan University",
"Kathmandu University");

    obj.displayList(universityList);
```

```
    List<String> programmingLanguages = Arrays.asList("Java", "C");
    obj.displayList(universityList, programmingLanguages);
  }
}
```

Warnings

```
Type safety: Potential heap pollution via varargs parameter lists
Type safety: A generic array of List<String> is created for a varargs
 parameter
```

Output

```
Note: Main.java uses unchecked or unsafe operations.
[Tribhuvan University, Kathmandu University]
[Tribhuvan University, Kathmandu University]
[Java, C]
```

Here, `List ... lists` specifies a variable-length argument of type `List`. This means that the method `displayList()` can have zero or more arguments.

The above program compiles without errors but gives warnings when `@SafeVarargs` annotation isn't used.

When we use `@SafeVarargs` annotation in the above example,

```
@SafeVarargs
```

```
 private void displayList(List<String>... lists) { ... }
```

We get the same output but without any warnings. Unchecked warnings are also suppressed when we use this annotation.

---

5. @FunctionalInterface

Java 8 first introduced this `@FunctionalInterface` annotation. This annotation indicates that the type declaration on which it is used is a functional interface. A functional interface can have only one abstract method.

Example 5: @FunctionalInterface annotation example

```
@FunctionalInterface

public interface MyFuncInterface{

  public void firstMethod(); // this is an abstract method

}
```

If we add another abstract method, let's say

```
@FunctionalInterface

public interface MyFuncInterface{

  public void firstMethod(); // this is an abstract method

  public void secondMethod(); // this throws compile error

}
```

Now, when we run the program, we will get the following warning:

```
Unexpected @FunctionalInterface annotation

@FunctionalInterface ^ MyFuncInterface is not a functional interface
```

```
multiple non-overriding abstract methods found in interface MyFuncInterface
```

It is not mandatory to use `@FunctionalInterface` annotation. The compiler will consider any interface that meets the functional interface definition as a functional interface.

We use this annotation to make sure that the functional interface has only one abstract method.

However, it can have any number of default and static methods because they have an implementation.

```
@FunctionalInterface

public interface MyFuncInterface{

  public void firstMethod(); // this is an abstract method

  default void secondMethod() { ... }

  default void thirdMethod() { ... }

}
```

Custom Annotations

It is also possible to create our own custom annotations.

Its syntax is:

```
[Access Specifier] @interface<AnnotationName> {

  DataType <Method Name>() [default value];

}
```

Here is what you need to know about custom annotation:

- Annotations can be created by using `@interface` followed by the annotation name.

- The annotation can have elements that look like methods but they do not have an implementation.
- The default value is optional. The parameters cannot have a null value.
- The return type of the method can be primitive, enum, string, class name or array of these types.

Example 6: Custom annotation example

```
@interface MyCustomAnnotation {

  String value() default "default value";

}



class Main {

  @MyCustomAnnotation(value = "programiz")

  public void method1() {

    System.out.println("Test method 1");

  }



  public static void main(String[] args) throws Exception {

    Main obj = new Main();

    obj.method1();

  }

}
```

Output

```
Test method 1
```

Meta Annotations

Meta-annotations are annotations that are applied to other annotations.

1. @Retention

The `@Retention` annotation specifies the level up to which the annotation will be available.

Its syntax is:

```
@Retention(RetentionPolicy)
```

There are 3 types of retention policies:

- RetentionPolicy.SOURCE - The annotation is available only at the source level and is ignored by the compiler.
- RetentionPolicy.CLASS - The annotation is available to the compiler at compile-time, but is ignored by the Java Virtual Machine (JVM).
- RetentionPolicy.RUNTIME - The annotation is available to the JVM.

For example,

```
@Retention(RetentionPolicy.RUNTIME)

public @interface MyCustomAnnotation{ ... }
```

---

2. @Documented

By default, custom annotations are not included in the official Java documentation. To include our annotation in the Javadoc documentation, we use the `@Documented` annotation.

For example,

```
@Documented

public @interface MyCustomAnnotation{ ... }
```

---

3. @Target

We can restrict an annotation to be applied to specific targets using the `@Target` annotation.

Its syntax is:

```
@Target(ElementType)
```

The `ElementType` can have one of the following types:

Element Type	Target
`ElementType.ANNOTATION_TYPE`	Annotation type
`ElementType.CONSTRUCTOR`	Constructors
`ElementType.FIELD`	Fields
`ElementType.LOCAL_VARIABLE`	Local variables
`ElementType.METHOD`	Methods

`ElementType.PACKAGE`	Package
`ElementType.PARAMETER`	Parameter
`ElementType.TYPE`	Any element of class

For example,

```
@Target(ElementType.METHOD)

public @interface MyCustomAnnotation{ ... }
```

In this example, we have restricted the use of this annotation to methods only.

> Note: If the target type is not defined, the annotation can be used for any element.

---

4. @Inherited

By default, an annotation type cannot be inherited from a superclass. However, if we need to inherit an annotation from a superclass to a subclass, we use the `@Inherited` annotation.

Its syntax is:

```
@Inherited
```

For example,

```
@Inherited

public @interface MyCustomAnnotation { ... }
```

```
@MyCustomAnnotation

public class ParentClass{ ... }



public class ChildClass extends ParentClass { ... }
```

---

5. @Repeatable

An annotation that has been marked by `@Repeatable` can be applied multiple times to the same declaration.

```
@Repeatable(Universities.class)

public @interface University {

  String name();

}
```

The value defined in the `@Repeatable` annotation is the container annotation. The container annotation has a variable `value` of array type of the above repeatable annotation. Here, `Universities` are the containing annotation type.

```
public @interface Universities {

  University[] value();

}
```

Now, the `@University` annotation can be used multiple times on the same declaration.

```
@University(name = "TU")

@University(name = "KU")
```

```
private String uniName;
```

If we need to retrieve the annotation data, we can use the Reflection API.

To retrieve annotation values, we use `getAnnotationsByType()` or `getAnnotations()` method defined in the Reflection API.

SQL Query Practice

Consider the below two tables for reference while trying to solve the **SQL queries for practice**.

Table – EmployeeDetails

EmpId	FullName	ManagerId	DateOfJoining	City
121	John Snow	321	01/31/2014	Toronto
321	Walter White	986	01/30/2015	California
421	Kuldeep Rana	876	27/11/2016	New Delhi

Table – EmployeeSalary

EmpId	Project	Salary	Variable
121	P1	8000	500
321	P2	10000	1000
421	P1	12000	0

Easy Questions

Ques.1. Write an SQL query to fetch the EmpId and FullName of all the employees working under Manager with id – '986'.

Ans. We can use the EmployeeDetails table to fetch the employee details with a where clause for the manager-

```
SELECT  EmpId, FullName
FROM EmployeeDetails
WHERE ManagerId = 986;
```

Ques.2. Write an SQL query to fetch the different projects available from the EmployeeSalary table.

Ans. While referring to the EmployeeSalary table, we can see that this table contains project values corresponding to each employee, or we can say that we will have duplicate project values while selecting Project values from this table.

So, we will use the distinct clause to get the unique values of the Project.

```
SELECT DISTINCT(Project)
FROM EmployeeSalary;
```

Ques.3. Write an SQL query to fetch the count of employees working in project 'P1'.

Ans. Here, we would be using aggregate function count() with the SQL **where** clause-

```
SELECT COUNT(*)
FROM EmployeeSalary
WHERE Project = 'P1';
```

Ques.4. Write an SQL query to find the maximum, minimum, and average salary of the employees.

Ans. We can use the aggregate function of SQL to fetch the max, min, and average values-

```
SELECT Max(Salary),
Min(Salary),
AVG(Salary)
FROM EmployeeSalary;
```

Ques.5. Write an SQL query to find the employee id whose salary lies in the range of 9000 and 15000.

Ans. Here, we can use the ‘Between’ operator with a where clause.

```
SELECT EmpId, Salary
FROM EmployeeSalary
WHERE Salary BETWEEN 9000 AND 15000;
```

Ques.6. Write an SQL query to fetch those employees who live in Toronto and work under manager with ManagerId – 321.

Ans. Since we have to satisfy both the conditions – employees living in ‘Toronto’ and working in Project ‘P2’. So, we will use AND operator here-

```
SELECT EmpId, City, ManagerId
FROM EmployeeDetails
WHERE City='Toronto' AND ManagerId='321';
```

Ques.7. Write an SQL query to fetch all the employees who either live in California or work under a manager with ManagerId – 321.

Ans. This interview question requires us to satisfy either of the conditions – employees living in ‘California’ and working under Manager with ManagerId ‘321’. So, we will use the OR operator here-

```
SELECT EmpId, City, ManagerId
FROM EmployeeDetails
WHERE City='California' OR ManagerId='321';
```

Ques.8. Write an SQL query to fetch all those employees who work on Project other than P1.

Ans. Here, we can use the NOT operator to fetch the rows which are not satisfying the given condition.

```
SELECT EmpId
FROM EmployeeSalary
WHERE NOT Project='P1';
```

Or using the not equal to operator-

```
SELECT EmpId
FROM EmployeeSalary
WHERE Project <> 'P1';
```

Hee khalcha difference related ahe warchay query shi , just read kaar

the difference between NOT and <> SQL operators

Ans:
NOT *negates* the following condition so it can be used with various operators. != is the non-standard alternative for the <> operator which means "not equal".
e.g.

```
NOT (a LIKE 'foo%')
```

```
NOT ( (a,b) OVERLAPS (x,y) )
```

```
 NOT (a BETWEEN x AND y)
```

```
NOT (a IS NULL)
```

Except for the overlaps operator above could also be written as:

```
a NOT LIKE 'foo%'
```

```
a NOT BETWEEN x AND y
```

```
a IS NOT NULL
```

In some situations it might be easier to understand to negate a complete expression rather then rewriting it to mean the opposite.

NOT *can* however be used with <> - but that wouldn't make much sense though: NOT (a <> b) is the same as a = b. Similarly you could use NOT to negate the equality operator NOT (a = b) is the same as a <> b

Ques.9. Write an SQL query to display the total salary of each employee adding the Salary with Variable value.

Ans. Here, we can simply use the '+' operator in SQL.

```
SELECT EmpId,
Salary+Variable as TotalSalary
FROM EmployeeSalary;
```

Ques.10. Write an SQL query to fetch the employees whose name begins with any two characters, followed by a text “hn” and ending with any sequence of characters.

Ans. For this question, we can create an SQL query using like operator with ‘_’ and ‘%’ wild card characters, where ‘_’ matches a single character and ‘%’ matches ‘0 or multiple characters’.

```
SELECT FullName
FROM EmployeeDetails
WHERE FullName LIKE ‘__hn%’;
```

Ques.11. Write an SQL query to fetch all the EmpIds which are present in either of the tables – ‘EmployeeDetails’ and ‘EmployeeSalary’.

Ans. In order to get unique employee ids from both the tables, we can use Union clause which can combine the results of the two SQL queries and return unique rows.

```
SELECT EmpId FROM EmployeeDetails
UNION
SELECT EmpId FROM EmployeeSalary;
```

Ques.12. Write an SQL query to fetch common records between two tables.

Ans. SQL Server – Using INTERSECT operator-

```
SELECT * FROM EmployeeSalary
INTERSECT
SELECT * FROM ManagerSalary;
```

MySQL – Since MySQL doesn’t have INTERSECT operator so we can use the sub query-

```
SELECT *
FROM EmployeeSalary
WHERE EmpId IN
(SELECT EmpId from ManagerSalary);
```

Ques.13. Write an SQL query to fetch records that are present in one table but not in another table.

Ans. SQL Server – Using MINUS- operator-

```
SELECT * FROM EmployeeSalary
MINUS
SELECT * FROM ManagerSalary;
```

MySQL – Since MySQL doesn't have MINUS operator so we can use LEFT join-

```
SELECT EmployeeSalary.*
FROM EmployeeSalary
LEFT JOIN
ManagerSalary USING (EmpId)
WHERE ManagerSalary.EmpId IS NULL;
```

Ques.14. Write an SQL query to fetch the EmpIds that are present in both the tables – 'EmployeeDetails' and 'EmployeeSalary.

Ans. Using sub query-

```
SELECT EmpId FROM

EmployeeDetails
where EmpId IN
(SELECT EmpId FROM EmployeeSalary);
```

Ques.15. Write an SQL query to fetch the EmpIds that are present in EmployeeDetails but not in EmployeeSalary.

Ans. Using sub query-

```
SELECT EmpId FROM
EmployeeDetails
where EmpId Not IN
(SELECT EmpId FROM EmployeeSalary);
```

Ques.16. Write an SQL query to fetch the employee full names and replace the space with '-'.

Ans. Using 'Replace' function-

```
SELECT REPLACE(FullName, ' ', '-')
FROM EmployeeDetails;
```

Ques.17. Write an SQL query to fetch the position of a given character(s) in a field.

Ans. Using 'Instr' function-

```
SELECT INSTR(FullName, 'Snow')
FROM EmployeeDetails;
```

Ques.18. Write an SQL query to display both the EmpId and ManagerId together.

Ans. Here we can use the CONCAT command.

```
SELECT CONCAT(EmpId, ManagerId) as NewId
FROM EmployeeDetails;
```

Ques.19. Write a query to fetch only the first name(string before space) from the FullName column of the EmployeeDetails table.

Ans. In this question, we are required to first fetch the location of the space character in the FullName field and then extract the first name out of the FullName field.

For finding the location we will use the LOCATE method in MySQL and CHARINDEX in SQL SERVER and for fetching the string before space, we will use the SUBSTRING OR MID method.

MySQL – using MID

```
SELECT MID(FullName, 1, LOCATE(' ',FullName))
FROM EmployeeDetails;
```

SQL Server – using SUBSTRING

```
SELECT SUBSTRING(FullName, 1, CHARINDEX(' ',FullName))

FROM EmployeeDetails;
```

Ques.20. Write an SQL query to upper case the name of the employee and lower case the city values.

Ans. We can use SQL Upper and Lower functions to achieve the intended results.

```
SELECT UPPER(FullName), LOWER(City)
FROM EmployeeDetails;
```

Ques.21. Write an SQL query to find the count of the total occurrences of a particular character – ‘n’ in the FullName field.

Ans. Here, we can use the ‘Length’ function. We can subtract the total length of the FullName field with a length of the FullName after replacing the character – ‘n’.

```
SELECT FullName,
LENGTH(FullName) - LENGTH(REPLACE(FullName, 'n', ''))
FROM EmployeeDetails;
```

Ques.22. Write an SQL query to update the employee names by removing leading and trailing spaces.

Ans. Using the ‘Update’ command with the ‘LTRIM’ and ‘RTRIM’ function.

```
UPDATE EmployeeDetails
SET FullName = LTRIM(RTRIM(FullName));
```

Ques.23. Fetch all the employees who are not working on any project.

Ans. This is one of the very basic interview questions in which the interviewer wants to see if the person knows about the commonly used – Is NULL operator.

```
SELECT EmpId
FROM EmployeeSalary
WHERE Project IS NULL;
```

Ques.24. Write an SQL query to fetch employee names having a salary greater than or equal to 5000 and less than or equal to 10000.

Ans. Here, we will use BETWEEN in the ‘where’ clause to return the EmpId of the employees with salary satisfying the required criteria and then use it as subquery to find the fullName of the employee from EmployeeDetails table.

```
SELECT FullName
FROM EmployeeDetails
WHERE EmpId IN
(SELECT EmpId FROM EmployeeSalary
WHERE Salary BETWEEN 5000 AND 10000);
```

Ques.25. Write an SQL query to find the current date-time.

Ans. MySQL-SELECT NOW();

SQL Server-SELECT getdate();

Oracle-SELECT SYSDATE FROM DUAL;

Ques.26. Write an SQL query to fetch all the Employees details from EmployeeDetails table who joined in the Year 2020.

Ans. Using BETWEEN for the date range ’01-01-2020′ AND ’31-12-2020′-

```
SELECT * FROM EmployeeDetails
WHERE DateOfJoining BETWEEN '2020/01/01'
AND '2020/12/31';
```

Also, we can extract year part from the joining date (using YEAR in mySQL)-

```
SELECT * FROM EmployeeDetails
WHERE YEAR(DateOfJoining) = '2020';
```

Ques.27. Write an SQL query to fetch all employee records from EmployeeDetails table who have a salary record in EmployeeSalary table.

Ans. Using ‘Exists’-

```
SELECT * FROM EmployeeDetails E
WHERE EXISTS
(SELECT * FROM EmployeeSalary S
WHERE  E.EmpId = S.EmpId);
```

Ques.28. Write an SQL query to fetch project-wise count of employees sorted by project’s count in descending order.

Ans. The query has two requirements – first to fetch the project-wise count and then to sort the result by that count.

For project-wise count, we will be using the GROUP BY clause and for sorting, we will use the ORDER BY clause on the alias of the project-count.

```
SELECT Project, count(EmpId) EmpProjectCount
FROM EmployeeSalary
GROUP BY Project
ORDER BY EmpProjectCount DESC;
```

Ques.29. Write a query to fetch employee names and salary records. Display the employee details even if the salary record is not present for the employee.

Ans. This is again one of the very common interview questions in which the interviewer just wants to check the basic knowledge of SQL JOINS.Here, we can use left join with EmployeeDetail table on the left side of the EmployeeSalary table.

```
SELECT E.FullName, S.Salary
FROM EmployeeDetails E
LEFT JOIN
EmployeeSalary S
ON E.EmpId = S.EmpId;
```

Ques.30. Write an SQL query to join 3 tables.

Ans. Considering 3 tables TableA, TableB, and TableC, we can use 2 joins clauses like below-


```
SELECT column1, column2
FROM TableA
JOIN TableB ON TableA.Column3 = TableB.Column3
JOIN TableC ON TableA.Column4 = TableC.Column4;
```

HARD Questions

Ques. 31. Write an SQL query to fetch all the Employees who are also managers from the EmployeeDetails table.

Ans. Here, we have to use Self-Join as the requirement wants us to analyze the EmployeeDetails table as two tables. We will use different aliases 'E' and 'M' for the same EmployeeDetails table.

```
SELECT DISTINCT E.FullName
FROM EmployeeDetails E
INNER JOIN EmployeeDetails M
ON E.EmpID = M.ManagerID;
```

Ques.32. Write an SQL query to fetch duplicate records from EmployeeDetails (without considering the primary key – EmpId).

Ans. In order to find duplicate records from the table, we can use GROUP BY on all the fields and then use the HAVING clause to return only those fields whose count is greater than 1 i.e. the rows having duplicate records.

```
SELECT FullName, ManagerId, DateOfJoining, City, COUNT(*)

FROM EmployeeDetails

GROUP BY FullName, ManagerId, DateOfJoining, City

HAVING COUNT(*) > 1;
```

Ques.33. Write an SQL query to remove duplicates from a table without using a temporary table.

Ans. Here, we can use delete with alias and inner join. We will check for the equality of all the matching records and them remove the row with higher EmpId.

```
DELETE E1 FROM EmployeeDetails E1

INNER JOIN EmployeeDetails E2

WHERE E1.EmpId > E2.EmpId

AND E1.FullName = E2.FullName

AND E1.ManagerId = E2.ManagerId

AND E1.DateOfJoining = E2.DateOfJoining

AND E1.City = E2.City;
```

Ques.34. Write an SQL query to fetch only odd rows from the table.

Ans. In case we have an auto-increment field e.g. EmpId then we can simply use the below query-

```
SELECT * FROM EmployeeDetails

WHERE MOD (EmpId, 2) <> 0;
```

In case we don't have such a field then we can use the below queries.

Using Row_number in SQL server and checking that the remainder when divided by 2 is 1-

```
SELECT E.EmpId, E.Project, E.Salary
FROM (
    SELECT *, Row_Number() OVER(ORDER BY EmpId) AS RowNumber
    FROM EmployeeSalary
) E
WHERE E.RowNumber % 2 = 1;
```

Using a user defined variable in MySQL-

```
SELECT *
FROM (
     SELECT *, @rowNumber := @rowNumber+ 1 rn
     FROM EmployeeSalary
     JOIN (SELECT @rowNumber:= 0) r
    ) t
WHERE rn % 2 = 1;
```

Ques.35. Write an SQL query to fetch only even rows from the table.

Ans. In case we have an auto-increment field e.g. EmpId then we can simply use the below query-

```
SELECT * FROM EmployeeDetails
WHERE MOD (EmpId, 2) = 0;
```

In case we don't have such a field then we can use the below queries.

Using Row_number in SQL server and checking that the remainder when divided by 2 is 1-

```
SELECT E.EmpId, E.Project, E.Salary
FROM (
   SELECT *, Row_Number() OVER(ORDER BY EmpId) AS RowNumber
   FROM EmployeeSalary
) E
WHERE E.RowNumber % 2 = 0;
```

Using a user defined variable in MySQL-

```
SELECT *
FROM (
    SELECT *, @rowNumber := @rowNumber+ 1 rn
```

```
    FROM EmployeeSalary
    JOIN (SELECT @rowNumber:= 0) r
   ) t
WHERE rn % 2 = 0;
```

Ques.36. Write an SQL query to create a new table with data and structure copied from another table.

Ans.

```
CREATE TABLE NewTable
SELECT * FROM EmployeeSalary;
```

Ques.37. Write an SQL query to create an empty table with the same structure as some other table.

Ans. Here, we can use the same query as above with False ‘WHERE’ condition-

```
CREATE TABLE NewTable
SELECT * FROM EmployeeSalary where 1=0;
```

Ques.38. Write an SQL query to fetch top n records?

Ans. In MySQL using LIMIT-

```
SELECT *

FROM EmployeeSalary

ORDER BY Salary DESC LIMIT N;
```

In SQL server using TOP command-

```
SELECT TOP N *

FROM EmployeeSalary

ORDER BY Salary DESC;
```

Ques.39. Write an SQL query to find the nth highest salary from table.

Ans, Using Top keyword (SQL Server)-

```
SELECT TOP 1 Salary
FROM (
    SELECT DISTINCT TOP N Salary
    FROM Employee
    ORDER BY Salary DESC
    )
ORDER BY Salary ASC;
```

Using limit clause(MySQL)-

```
SELECT Salary
FROM Employee
ORDER BY Salary DESC LIMIT N-1,1;
```

Ques.40. Write SQL query to find the 3rd highest salary from a table without using the TOP/limit keyword.

Ans. This is one of the most commonly asked interview questions. For this, we will use a correlated subquery.

In order to find the 3rd highest salary, we will find the salary value until the inner query returns a count of 2 rows having the salary greater than other distinct salaries.

```
SELECT Salary
FROM EmployeeSalary Emp1
WHERE 2 = (
            SELECT COUNT( DISTINCT ( Emp2.Salary ) )
            FROM EmployeeSalary Emp2
            WHERE Emp2.Salary > Emp1.Salary
        )
```

For nth highest salary-

```
SELECT Salary
FROM EmployeeSalary Emp1
WHERE N-1 = (
            SELECT COUNT( DISTINCT ( Emp2.Salary ) )
            FROM EmployeeSalary Emp2
            WHERE Emp2.Salary > Emp1.Salary
        )
```

Java Sorting Algorithms

for all sorting algorithms in Java:

Algorithm	Approach	Best Time Complexity
Merge Sort	Split the array into smaller subarrays till pairs of elements are achieved, and then combine them in such a way that they are in order.	O(n log (n))
Heap Sort	Build a max (or min) heap and extract the first element of the heap (or root), and then send it to the end of the heap. Decrement the size of the heap and repeat till the heap has only one node.	O(n log (n))
Insertion Sort	In every run, compare it with the predecessor. If the current element is not in the correct location, keep shifting the predecessor subarray till the correct index for the element is found.	O (n)
Selection Sort	Find the minimum element in each run of the array and swap it with the element at the current index is compared.	O(n^2)
Bubble Sort	Keep swapping elements that are not in their right location till the array is sorted.	O (n)

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)