Q1. What is Exception in Java?

Answer: An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner..

Q2. What is Exception Handling?

Answer: Exception Handling in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.
Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. 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 the state of the program when the exception occurred.

Major reasons why an exception Occurs.

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

Q3. What is the difference between Checked and Unchecked Exceptions and Error?

Answer:

- **Checked exceptions** − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a FileNotFoundException occurs, and the compiler prompts the programmer to handle the exception.

Example:

```
import java.io.File;

import java.io.FileReader;


public class FilenotFound_Demo {


  public static void main(String args[]) {

    File file = new File("E://file.txt");

    FileReader fr = new FileReader(file);

  }
```

```
}
```

If we try to compile the above program, we will get the following exceptions.

Output

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or
declared to be thrown
    FileReader fr = new FileReader(file);
                    ^
1 error
```

Note − Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

- **Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- For example, if we have declared an array of size 5 in your program, and trying to call the 6th element of the array then an ArrayIndexOutOfBoundsExceptionexception occurs.
  Example

```
public class Unchecked_Demo {

  public static void main(String args[]) {
    int num[] = {1, 2, 3, 4};
    System.out.println(num[5]);
  }
}
```

If you compile and execute the above program, you will get the following exception.

Output

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

- **Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Q4. What are the difference between throw and throws in Java?

Answer: In Java, throw and throws are related to exception handling, but they have different purposes and usage.

throw:

- The throw keyword is used to explicitly throw an exception within a method or block of code.
- It is used to raise an exception when a certain condition or situation occurs that requires abnormal termination of the program flow.
- When you throw an exception using throw, you specify the exception object that should be thrown.
- The thrown exception must be a subclass of Throwable (either a subclass of Exception or Error).

Example:

```
public void divide(int dividend, int divisor) {

  if (divisor == 0) {

    throw new ArithmeticException("Divisor cannot be zero.");
```

```
    }
    // Perform division if divisor is non-zero
  }
```

throws:

- The throws keyword is used in method declarations to indicate that the method may throw one or more types of exceptions.
- It specifies the exception types that the method might throw but doesn't handle within the method itself.
- The exceptions listed with throws must be checked exceptions (subclasses of Exception, excluding RuntimeException and its subclasses).
- It informs the caller of the method that they need to handle or propagate the listed exceptions.

Example:

```
public void readFile() throws IOException {
  // Code that may throw IOException
}
```

Q5. What is multithreading in Java? mention its advantages?

Answer: Multithreading in Java refers to the concurrent execution of multiple threads within a Java program. A thread is a lightweight unit of execution that can run concurrently with other threads, allowing multiple tasks to be performed simultaneously.

Advantages of multithreading in Java:

- **Concurrency and Responsiveness:** Multithreading enables the execution of multiple tasks concurrently, allowing the program to perform multiple operations simultaneously. This can lead to improved responsiveness and enhanced user experience, particularly in applications that involve time-consuming or blocking operations such as network communication or file I/O.

- **Utilization of CPU Cores:** In a multicore or multiprocessor system, multithreading allows the program to utilize multiple CPU cores effectively. By dividing the workload among multiple threads, the program can take advantage of the available processing power, leading to improved performance and faster execution times.

- **Improved Efficiency:** Multithreading can improve the overall efficiency of a program by maximizing resource utilization. While one thread is waiting for an I/O operation or a blocking task to complete, another thread can continue executing, utilizing the CPU cycles that would otherwise be idle.

- **Simplified Design:** Multithreading allows for the design of more efficient and streamlined programs. Complex tasks can be divided into smaller, manageable threads, each responsible for a specific aspect of the task. This modular approach simplifies the design, maintenance, and testing of the code.

- **Asynchronous Programming:** Multithreading facilitates asynchronous programming, where tasks can be initiated and run independently in the background. This is especially useful in scenarios where certain tasks can proceed without waiting for others to complete, leading to improved performance and responsiveness.

- **Parallel Computing**: With the availability of multiple threads, computationally intensive tasks can be divided into smaller parts and executed in parallel, potentially reducing the overall execution time. This is particularly beneficial in applications involving heavy computations, such as scientific simulations or data processing.

It's important to note that multithreading also introduces challenges such as thread synchronization, resource sharing, and potential concurrency issues. Proper synchronization and coordination mechanisms need to be implemented to ensure thread safety and prevent race conditions.

Q6. Write a program to create and call a custom exception.

Answer:

```
// Custom exception class
class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }
}

// Example class that throws the custom exception
class CustomExceptionExample {
    public static void divide(int dividend, int divisor) throws MyCustomException {
        if (divisor == 0) {
            throw new MyCustomException("Divisor cannot be zero.");
        }
        int result = dividend / divisor;
        System.out.println("Result: " + result);
    }
}

// Main class to demonstrate the usage
public class Main {
    public static void main(String[] args) {
        try {
            CustomExceptionExample.divide(10, 2); // No exception thrown
            CustomExceptionExample.divide(10, 0); // Throws MyCustomException
        } catch (MyCustomException e) {
            System.out.println("Custom exception caught: " + e.getMessage());
```

```
        }
    }
}
```

in the above example, we create a custom exception class called MyCustomException, which extends the Exception class. It has a constructor that accepts a message to provide more information about the exception.

The CustomExceptionExample class contains a divide method that throws the custom exception if the divisor is zero. Otherwise, it performs the division and prints the result.

In the Main class, we demonstrate the usage of the custom exception by calling the divide method within a try-catch block. If the exception is thrown, we catch it and print the custom exception message.

Output of program:

Result: 5

Custom exception caught: Divisor cannot be zero.

Q7. How can you handle exceptions in Java?

Answer: In Java, exceptions are used to handle exceptional situations or errors that may occur during the execution of a program. To handle exceptions, you can use try-catch blocks. The try block contains the code that may throw an exception, and the catch block handles the exception if it occurs. Here's a general structure:

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 exception1) {
    // Handle exception1
} catch (ExceptionType2 exception2) {
    // Handle exception2
} finally {
    // Optional: Code that will always execute, regardless of whether an exception occurred or not
}
```

Let's break down each part:

try: The try block encloses the code that might throw an exception. If an exception occurs within this block, it is immediately caught and handled in one of the catch blocks.

catch: The catch block specifies the type of exception it can handle, denoted by ExceptionType. When an exception of that type (or one of its subclasses) is thrown, the catch block is executed to handle the exception. You can have multiple catch blocks to handle different types of exceptions.

finally: The finally block is optional and follows the try-catch blocks. It contains code that will always execute, regardless of whether an exception occurred or not. It's typically used for cleanup tasks, such as releasing resources.

Example:

```
try {
```

```
    // Code that may throw an exception
    int result = divide(10, 0);
    System.out.println("Result: " + result);
} catch (ArithmeticException ex) {
    // Handle the arithmetic exception
    System.out.println("Cannot divide by zero!");
} finally {
    // Cleanup code
    System.out.println("End of execution");
}

// Method that throws an exception
public static int divide(int dividend, int divisor) {
    return dividend / divisor;
}
```

In this example, the divide method throws an ArithmeticException if the divisor is zero. The catch block catches the exception and handles it by printing an error message. The finally block executes regardless of whether an exception occurred or not, in this case, printing "End of execution".

Q8. What is Thread in Java?

Answer: A Thread is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform complicated tasks in the background, we used the Thread concept in Java. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.

Q9. What are the two ways of implementing thread in Java?

Answer: There are 2 different ways to create Threads in Java.

- By extending Thread class
- By implementing Runnable interface

By extending Thread class

First, let’s look at the syntax of creating a Thread

```
public class Main extends Thread {
        public void run () {
                System.out.println("This code is running in a thread");
        }
}
```

In order the create Thread we need to extend a Main class with the Thread class. Once we extend the Thread class, we can able to override the run() method.

complete implementation –

```
public class Main extends Thread {

	public static void main(String[] args) {

		Main thread = new Main();

		thread.start();

		System.out.println("This code is running outside of the thread");

	}

	public void run () {

		System.out.println("This code is running in a thread");

	}

}
```

Inside the main(), we call the start() method to start the Thread

Output:

This code is running in a thread

This code is running outside of the thread

By implementing Runnable interface

First, let’s look at the syntax of creating Thread

```
public class Main implements Runnable {

	public void run () {

		System.out.println("This code is running in a thread");

	}

}
```

We need to implement the Main class with a Runnable interface. Once we implement the Runnable interface, we can able to override the run() method.

the complete implementation –

```
public class Main implements Runnable {

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

```
        Main obj = new Main();

        Thread thread = new Thread(obj);

        thread.start();

        System.out.println("This code is running outside of the thread");

    }

    public void run () {

        System.out.println("This code is running in a thread");

    }

}
```

Inside the main() method, we will create the Thread object using a new keyword, then on the thread object, we will call the start() method to start the Thread.

Output:

This code is running in a thread

This code is running outside of the thread

Q10.What do you mean by garbage collection?

Answer: Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.