Q1.What is Exception in Java?

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. It is a way to handle error conditions or exceptional situations that may arise while a program is running. When an exception occurs, it is said to be "thrown" by the code that encounters the exceptional condition.

Here's a real-life example to help illustrate the concept of exceptions in Java:

Imagine you are writing a program to read data from a file. You expect the file to exist and be accessible, but sometimes things can go wrong. Let's say the file is accidentally deleted or renamed before your program runs. In this case, when your program tries to open the file, an exception will be thrown because the file is not found.

To handle this exception, you can use a try-catch block in your code. The try block contains the code that may throw an exception, and the catch block is used to handle the exception. Here's an example:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileReadExample {
    public static void main(String[] args) {
        try {
            File file = new File("example.txt");
            Scanner scanner = new Scanner(file);
            // Code to read data from the file
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
            // Code to handle the exception
        }
    }
}
```

Q2.What is Exception Handling?

Exception handling is a mechanism in programming languages, including Java, that allows you to gracefully handle and recover from exceptional situations or errors that may occur during the execution of a program. It provides a structured way to deal with exceptions and maintain the normal flow of the program.

Exception handling involves three main components:

Try Block: The code that may throw an exception is placed inside a try block. It is a block of code where you anticipate an exception to occur. If an exception is thrown within the try block, it is immediately caught and handled.

Catch Block: A catch block follows the try block and is used to catch and handle the thrown exception. It specifies the type of exception it can handle. If the exception matches the specified type, the catch block is executed.

Finally Block (Optional): The finally block is an optional block that follows the catch block (or the try block if no catch block is present). It is executed regardless of whether an exception occurred or not. The finally block is commonly used to perform cleanup operations, such as closing files or releasing resources, that need to be executed regardless of exceptions.

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

Exceptions are categorized into three main types: checked exceptions, unchecked exceptions, and errors. Here's a breakdown of the differences between these three categories:

Checked Exceptions:

- Checked exceptions are exceptions that are checked at compile-time by the Java compiler.
- They extend the Exception class (or one of its subclasses) but not the RuntimeException class.
- Examples of checked exceptions include IOException, SQLException, and ClassNotFoundException.

Unchecked Exceptions:

- Unchecked exceptions are exceptions that are not checked at compile-time.
- They extend the RuntimeException class or one of its subclasses.
- Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, and IllegalArgumentException.
- Unchecked exceptions do not need to be declared in the method signature or caught explicitly. The programmer has the choice to catch them or let them propagate up the call stack.

Errors:

- Errors are exceptional conditions that occur at runtime and are typically beyond the programmer's control.
- Errors extend the Error class and its subclasses.
- Examples of errors include OutOfMemoryError, StackOverflowError, and VirtualMachineError.

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

"Throw" and "Throws" are both used in exception handling, but they serve different purposes. Here's the difference between the two:

Throw:

- The "throw" keyword is used to explicitly throw an exception within a method.
- It is used when you encounter an exceptional condition or error within the code and want to manually create and throw an exception to signal the occurrence of that condition.
- The "throw" keyword is followed by an instance of an exception class or a subclass of the Exception class.
- It is typically used within a method body to indicate that an exceptional situation has occurred, and the execution of the method cannot proceed normally.
- When a method throws an exception using "throw", it is the responsibility of the caller or the calling method to handle the thrown exception.

throws:

- The "throws" keyword is used in a method declaration to indicate that the method may throw one or more exceptions.
- It specifies the type of exceptions that the method might throw during its execution.
- The "throws" keyword is followed by the names of the exception classes, separated by commas.
- It is used to delegate the responsibility of handling exceptions to the calling method or to allow exceptions to propagate up the call stack.
- When a method declares "throws" for a particular exception, it is indicating that it does not handle the exception itself, and the caller of that method is responsible for handling the exception.

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

Multithreading in Java refers to the concurrent execution of multiple threads within a single program. A thread is a lightweight unit of execution that can run concurrently with other threads, allowing for parallel or concurrent processing. Multithreading enables a program to perform multiple tasks concurrently, improving performance and responsiveness.

Advantages of multithreading in Java:

Increased responsiveness: Multithreading allows a program to remain responsive even while performing time-consuming operations. By executing tasks concurrently in separate threads, the program can continue to respond to user input or perform other operations without being blocked.
Improved performance: Multithreading can improve the overall performance of a program, especially on systems with multiple processors or CPU cores. By utilizing multiple threads, the program can take advantage of the available processing power to execute tasks in parallel, potentially reducing the total execution time.
Efficient resource utilization: Multithreading enables efficient utilization of system resources. By dividing tasks into separate threads, the program can efficiently utilize CPU time, memory, and other resources. This allows for better resource management and maximizes the utilization of available system resources.
Simplified program design: Multithreading allows for a more modular and organized program design. By dividing tasks into separate threads, each responsible for a specific part of the program's functionality, the code becomes more manageable and easier to maintain. It allows for better separation of concerns and promotes code reusability.

Concurrency and responsiveness in GUI applications: Multithreading is particularly useful in graphical user interface (GUI) applications. By running time-consuming tasks in separate threads, the GUI remains responsive and doesn't freeze or become unresponsive while performing those tasks. This enhances the user experience by providing a smooth and interactive interface.

Parallel processing: Multithreading enables parallel processing, which is crucial for computationally intensive tasks. By dividing a large task into smaller subtasks and executing them concurrently in separate threads, the program can achieve parallelism and potentially speed up the execution of the task.

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

```
class MyCustomException extends Exception {
 public MyCustomException(String message) {

super(message)
  }
}

public class CustomExceptionExample {

public static void main(String[] args) {
        try {
            int age = 15;
            if (age < 18) {
                throw new MyCustomException("You must be at least
18 years old.");
            } else {
                System.out.println("Access granted. You are eligible.");
            }
        } catch (MyCustomException e) {
            System.out.println("Exception caught: " +
e.getMessage());
        }
    }
}
```

Q7.How can you handle exceptions in Java?

Exceptions can be handled using exception handling mechanisms. The primary mechanisms for handling exceptions are:

we can enclose the code that may throw an exception within a try block.

- Immediately following the try block, you can include one or more catch blocks that specify the type of exception to catch and the corresponding code to handle the exception.
- If an exception occurs within the try block, the catch block that matches the exception type will be executed, allowing you to handle the exception gracefully.
- You can have multiple catch blocks to handle different types of exceptions or to provide different handling logic for specific exceptions.

finally blocks:

- A finally block can be used to specify code that must be executed regardless of whether an exception occurred or not.
- The finally block is optional and follows the catch blocks (if present) or the try block (if no catch block is used).
-
- It is typically used to perform cleanup operations, such as closing resources, releasing locks, or finalizing operations, that need to be executed regardless of exceptions.

Q8.What is Thread in Java?

A thread refers to a lightweight unit of execution that enables concurrent programming. A thread represents an independent path of execution within a program, allowing multiple tasks to run concurrently. Threads allow for parallel or concurrent execution of code, which can improve performance and enable efficient utilization of system resources.

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

There are two main ways to implement threads:

Extending the Thread class:

- The first way is to create a subclass of the Thread class and override its run() method.
- By extending the Thread class, you can define your thread's behavior by implementing the run() method.
- The run() method contains the code that will be executed when the thread starts.
- To start the thread, you create an instance of your subclass and call the start() method on that instance.

Implementing the Runnable interface:

- The second way is to implement the Runnable interface and pass an instance of the implementing class to the Thread constructor.
- The Runnable interface defines a single method, run(), which represents the code to be executed by the thread.
- Implementing the Runnable interface is often preferred as it allows better separation of concerns and supports better code reusability.

Q10.What do you mean by garbage collection?

Garbage collection in Java refers to the automatic memory management process where the Java Virtual Machine (JVM) automatically reclaims memory that is no longer in use by the program. It is responsible for identifying and freeing up memory occupied by objects that are no longer reachable or referenced by the program.

Some key points about garbage collection:

- Memory allocation in Java.
- Object reachability.
- Garbage collection process.