Java Exceptions

1. **Throwable Class Hierarchy:**
    - Throwable is the root class of the Java exception hierarchy. It has two main subclasses: Exception and Error.
    - Exception represents exceptional conditions that can be caught and handled within the program.
    - Error represents serious problems that are usually caused by external factors or issues within the JVM.
2. **Checked Exceptions and Unchecked Exceptions:**
    - Checked exceptions: These are exceptions that must be declared in the method signature or caught within the code. They extend the Exception class (excluding subclasses of RuntimeException). Examples include IOException, SQLException, and ClassNotFoundException.
    - Unchecked exceptions: These exceptions do not require explicit handling and are subclasses of RuntimeException. They often indicate programming errors or unexpected conditions. Examples include NullPointerException, ArrayIndexOutOfBoundsException, and ClassCastException.
3. **Exception Handling:**
    - try-catch blocks: Exceptions are caught and handled using try-catch blocks. The try block contains the code that may throw an exception, and the catch block specifies the code to be executed when a specific exception is caught.
    - Multiple catch blocks: Multiple catch blocks can be used to handle different types of exceptions. They are evaluated sequentially, and the first matching catch block is executed.
    - Optional finally block: The finally block, if present, is executed regardless of whether an exception is thrown or caught. It is typically used to release resources or perform cleanup operations.
4. **Exception Propagation:**
    - When an exception is thrown, it can be propagated up the call stack until it is caught and handled. If an exception is not caught, it will cause the program to terminate, printing a stack trace that shows the sequence of method calls that led to the exception.
5. **The throws Clause:**

- Checked exceptions that are not caught within a method must be declared in the method signature using the throws keyword. This informs the caller that the method may throw certain exceptions and that the caller must handle them accordingly.

6. **Custom Exceptions:**
    - Developers can create their own custom exceptions by extending the Exception class or its subclasses. This allows for the creation of application-specific exception types to represent unique error conditions.
    - Custom exceptions can have additional fields, constructors, and methods to provide specific information and behaviors.
7. **Exception Chaining:**
    - Exceptions can be chained together using the initCause() method. This allows one exception to be associated with another exception that caused it, providing more detailed error information.
8. **Try-with-resources:**
    - The try-with-resources statement simplifies the management of resources that need to be closed after use.
    - Resources that implement the AutoCloseable interface, such as file streams or database connections, can be declared within the try statement. These resources are automatically closed at the end of the block, ensuring proper cleanup, even in the presence of exceptions.
9. **Stack Trace and Error Messages:**
    - When an exception is thrown, a stack trace is generated, which provides information about the sequence of method calls leading to the exception. This information is useful for debugging purposes.
    - Exceptions can have error messages associated with them, which can be accessed using the getMessage() method. Error messages provide additional information about the exception and can be helpful in understanding the cause of the error.

The Java exception API provides a powerful mechanism for handling and managing exceptions, allowing developers to write robust and reliable code. It enables the identification and handling of exceptional conditions, facilitating graceful error recovery and enhancing program stability.

Writing a custom exception:

To write a custom exception in Java, you need to create a new class that extends either the Exception class or one of its subclasses. Here's a step-by-step guide:

1. **Choose the appropriate superclass**: Decide whether your custom exception should extend the Exception class or a more specific subclass, such as RuntimeException if it represents an unchecked exception.
2. **Create a new class:** Create a new class that inherits from the chosen superclass. Give it a meaningful name that reflects the specific error condition it represents. For example, if you are creating an exception for an invalid input, you could name it InvalidInputException.

```
public class InvalidInputException extends Exception {
   // Custom exception code goes here
}
```

3. **Provide constructors:** Define constructors to initialize the exception object. You can provide multiple constructors with different parameters to allow flexibility in creating the exception object.

```
public class InvalidInputException extends Exception {
   public InvalidInputException() {
      super();
   }

   public InvalidInputException(String message) {
      super(message);
   }

   public InvalidInputException(String message, Throwable cause) {
      super(message, cause);
   }
}
```

4. **Customize the exception behavior (optional):** You can add additional fields, methods, or behaviors to your custom exception to provide specific information or functionality. For example, you might add a method to retrieve additional details about the error condition.

```
public class InvalidInputException extends Exception {
    private int errorCode;

    public InvalidInputException(String message, int errorCode) {
        super(message);
        this.errorCode = errorCode;
    }

    public int getErrorCode() {
        return errorCode;
    }
}
```

5. **Throw and catch your custom exception:** You can throw your custom exception in appropriate places within your code when the specific error condition occurs. You can catch and handle the exception using try-catch blocks.

```
public class CustomExceptionExample {
    public void validateInput(String input) throws InvalidInputException {
        if (input == null || input.isEmpty()) {
            throw new InvalidInputException("Invalid input provided.", 1001);
        }
        // Other validation logic
    }
    public static void main(String[] args) {
        CustomExceptionExample example = new CustomExceptionExample();
        try {
            example.validateInput(null);
        } catch (InvalidInputException e) {
```

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

In the above example, the validateInput method throws the InvalidInputException when an invalid input is detected. In the main method, the exception is caught, and the error message and error code are displayed.

By creating custom exceptions, you can handle specific error conditions in a more structured and meaningful way, making your code more maintainable and easier to understand.

Example of using the try-with-resources statement to automatically manage resources that need to be closed:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class TryWithResourcesExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.out.println("Error reading the file: " + e.getMessage());
        }
    }
}
```

In the above example, we're reading lines from a file using a BufferedReader. The try-with-resources statement is used to automatically close the BufferedReader after we finish using it. Here's how it works:

1. The **BufferedReader** is created within the parentheses of the try-with-resources statement. This ensures that the resource is properly initialized and available within the try block.
2. **Inside the try block**, we read lines from the file using the readLine() method of the BufferedReader in a loop until we reach the end of the file. Each line is printed to the console.
3. Once the try block is finished, the BufferedReader resource is automatically closed, even if an exception occurs within the try block. This ensures that the file is properly closed and any associated system resources are released.
4. If an IOException occurs while reading the file, the exception is caught in the catch block, and an error message is displayed.

By using the try-with-resources statement, we eliminate the need for an explicit finally block to handle resource cleanup. The resources declared within the try-with-resources statement are automatically closed in reverse order of their creation.

Note: The resources used with try-with-resources must implement the AutoCloseable interface, which includes many standard Java classes for handling I/O operations, such as streams and readers/writers. If you have a custom resource that needs to be automatically closed, you can implement the AutoCloseable interface in your class and provide the necessary close() method implementation.

Session programs:

```
package exception;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class CheckedExceptionDemo {

   public static void main(String[] args) {
      //checked exception
      //Try catch block is mandatory in checked
      //exception, compiler will give error if not
      //handled
      CheckedExceptionDemo checkedExceptionDemo
            = new CheckedExceptionDemo();
      try {
         System.out.println(Files.readAllLines(Paths.get("./testfile/test1.txt")));
      }catch (Exception e){
         System.out.println("File not exits, please add file or correct the path");
      }
```



```
    }
    //This method is to simulate checked
    // exception scenario
    private File readSomeFile() throws IOException {
        return new File("test");
    }
}

package exception;

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

        try {
            //unchecked exception
            //Try catch block is not mandatory in
            //unchecked exception
            int result = 1 / 0;
            System.out.println(result);
        }catch (Exception abc){
            System.out.println("Exception !! " +
                    "cannot divide by zero");
            System.out.println(abc.getMessage());
        }

        System.out.println("last statement of code");
    }
}
```