Exception Handling

CLO: 2

Objectives

- **Exception Handling**

Exception Handling

Exception handling mechanisms allow developers to write robust programs by catching and handling the exceptions gracefully. Exceptions should be used **only for exceptional conditions** and not for control flow.

Exception

- An **exception** is an event that occurs during the execution of a program that disrupts its normal flow.
- For example: division by zero, invalid input

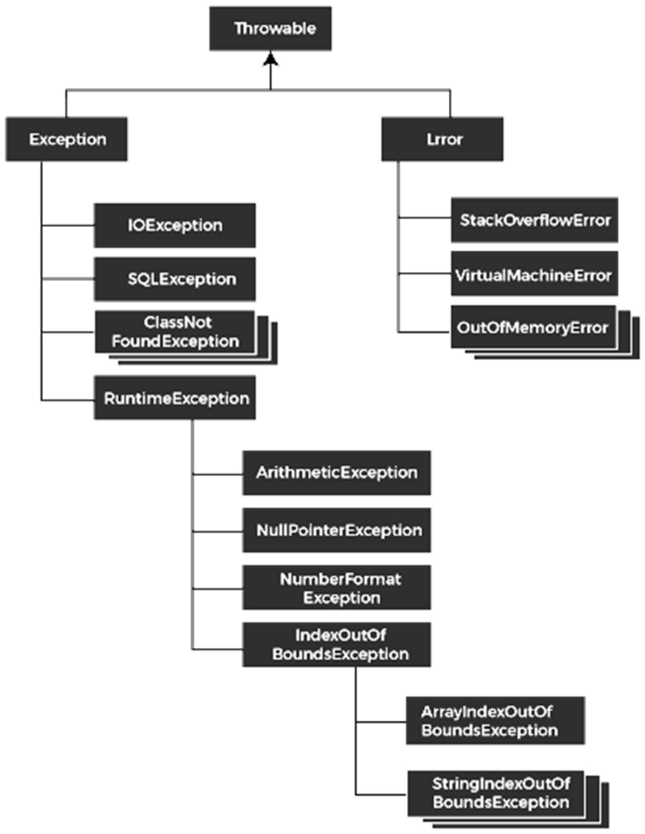
Types of Exceptions

Java has two main categories of exceptions:

1. **Checked Exceptions**: These exceptions are checked at compile-time. You must handle them explicitly by either using a try-catch block or declaring them with the throws keyword. Examples include:
    - **IOException**: Occurs when an I/O operation fails, like reading from a file that doesn't exist.
    - **SQLException**: Raised during SQL operations when there is a database access error.

2. **Unchecked Exceptions**: These are not checked at compile-time but are thrown during runtime. They usually occur due to programming errors like logical mistakes or incorrect use of APIs. Unchecked exceptions extend **RuntimeException**. Examples include:
    - **NullPointerException**: Thrown when an application attempts to use an object reference that has not been initialized.
    - ArrayIndexOutOfBoundsException: Raised when trying to access an invalid array index.

The Hierarchy of Java Exception classes is as follows



Exception Handling Keywords

Java provides several keywords to handle exceptions:

- **try**: The block where exceptions can occur.
- **catch**: The block that catches and handles the exception thrown in the try block.
- **finally**: A block that always executes, regardless of whether an exception occurred, used to clean up resources.
- **throw**: Used to explicitly throw an exception.
- **throws**: Used to declare an exception in the method signature, signaling that this method might throw an exception.

Best Practices for Exception Handling

1. **Use specific exceptions**: Catch specific exceptions like FileNotFoundException instead of generic ones like Exception.
2. **Clean-up in finally block**: Always release resources such as files or database connections in the finally block to ensure they are closed.
3. **Avoid empty catch blocks**: Always log or handle exceptions appropriately.
4. **Throw custom exceptions**: Use custom exceptions when specific scenarios need special handling.

Error

- Errors represent serious problems that cannot typically be handled by the application.
- For example: memory leaks or stack overflows
- Errors are a subclass of Throwable, but unlike exceptions, they are not meant to be caught.

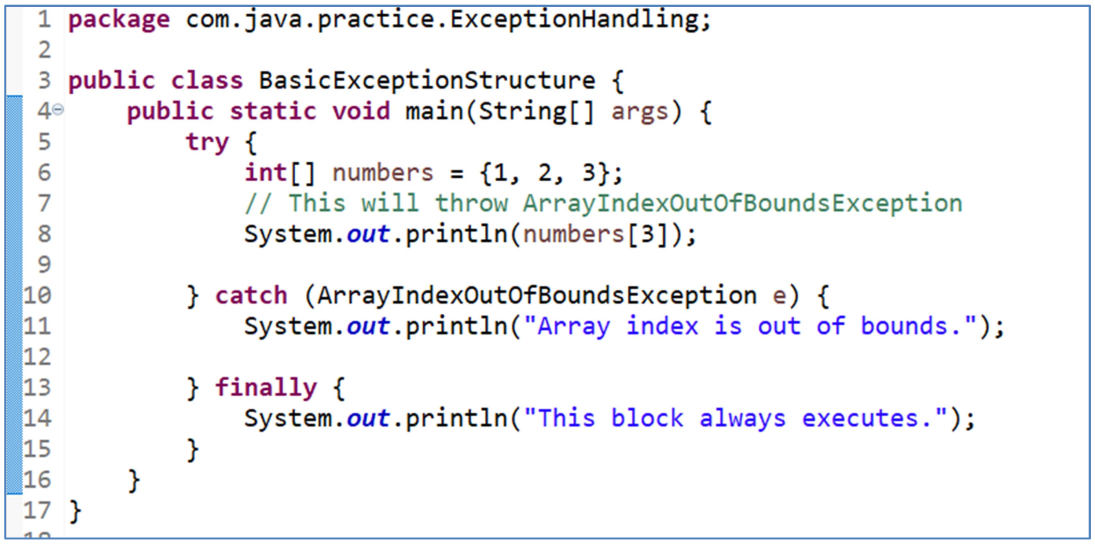
Exceptions VS Errors

Here are key differences between exception and error.

Feature	**Exception**	**Error**
Hierarchy	Exceptions belong to the Exception class hierarchy.	Errors belong to the Error class hierarchy.
Recoverable	Exceptions are generally recoverable and can be handled by the program.	Errors are typically not recoverable and should not be caught or handled by the program.
Type	Checked or unchecked (runtime) conditions that occur due to bad logic, user input, or other factors like file not found, invalid array index, etc.	Critical system-level issues like memory exhaustion, hardware failure, or JVM crashes.
Handling	Exceptions are meant to be caught and handled by the program using try-catch blocks.	Errors are not meant to be handled in the program since they represent critical issues that the program cannot fix.
Origin	Exceptions are usually caused by issues within the application logic or runtime environment.	Errors are generally caused by external issues related to the system or JVM, such as resource limitations.
Responsibility	It is the programmer's responsibility to handle exceptions appropriately.	Errors usually indicate system-level issues that cannot be addressed by the application code itself.

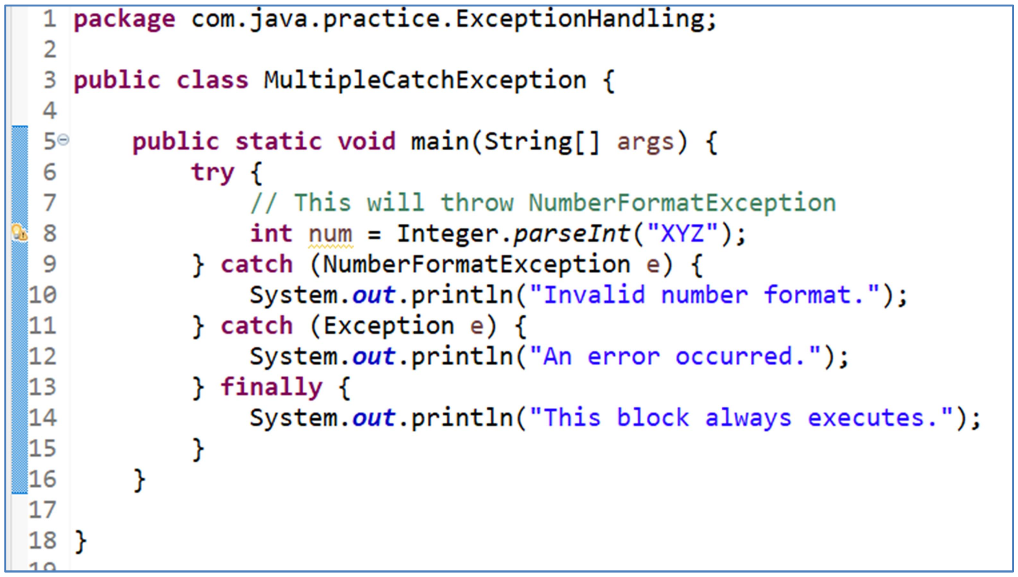
Examples

Example # 1:



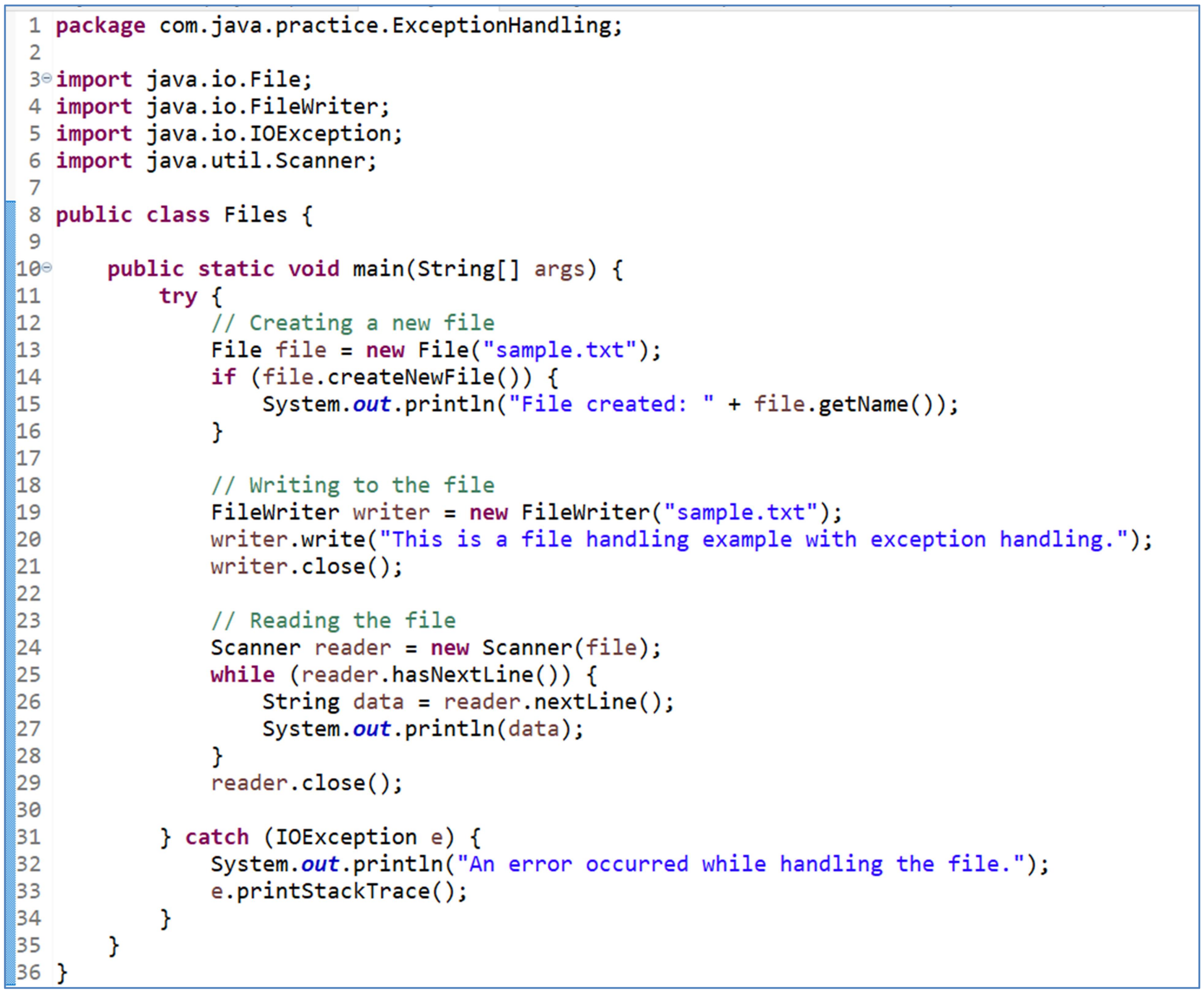
```java
package com.java.practice.ExceptionHandling;

public class BasicExceptionStructure {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            // This will throw ArrayIndexOutOfBoundsException
            System.out.println(numbers[3]);

        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index is out of bounds.");

        } finally {
            System.out.println("This block always executes.");
        }
    }
}
```

Example # 2:



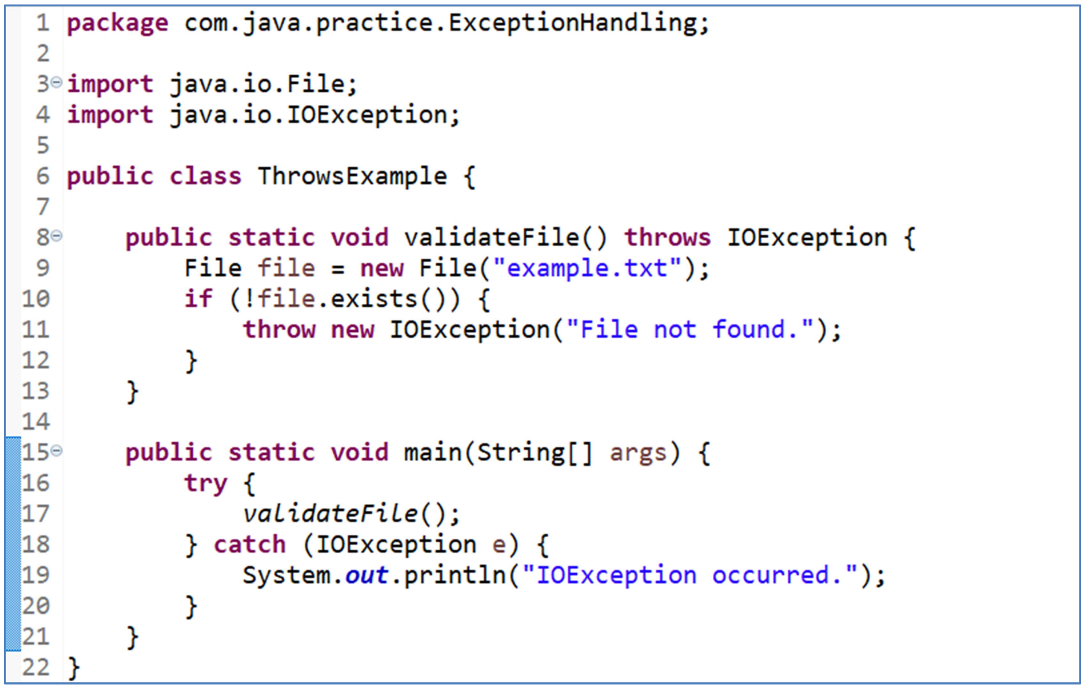
```java
package com.java.practice.ExceptionHandling;

public class MultipleCatchException {

    public static void main(String[] args) {
        try {
            // This will throw NumberFormatException
            int num = Integer.parseInt("XYZ");
        } catch (NumberFormatException e) {
            System.out.println("Invalid number format.");
        } catch (Exception e) {
            System.out.println("An error occurred.");
        } finally {
            System.out.println("This block always executes.");
        }
    }

}
```

Example # 3:



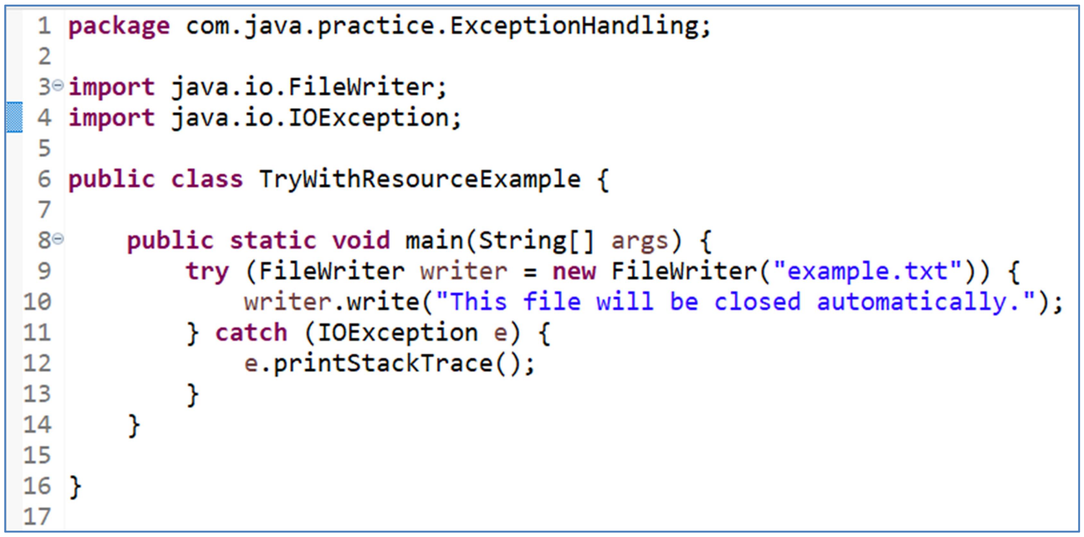
```java
package com.java.practice.ExceptionHandling;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class Files {

    public static void main(String[] args) {
        try {
            // Creating a new file
            File file = new File("sample.txt");
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            }

            // Writing to the file
            FileWriter writer = new FileWriter("sample.txt");
            writer.write("This is a file handling example with exception handling.");
            writer.close();

            // Reading the file
            Scanner reader = new Scanner(file);
            while (reader.hasNextLine()) {
                String data = reader.nextLine();
                System.out.println(data);
            }
            reader.close();

        } catch (IOException e) {
            System.out.println("An error occurred while handling the file.");
            e.printStackTrace();
        }
    }
}
```

Example # 4:



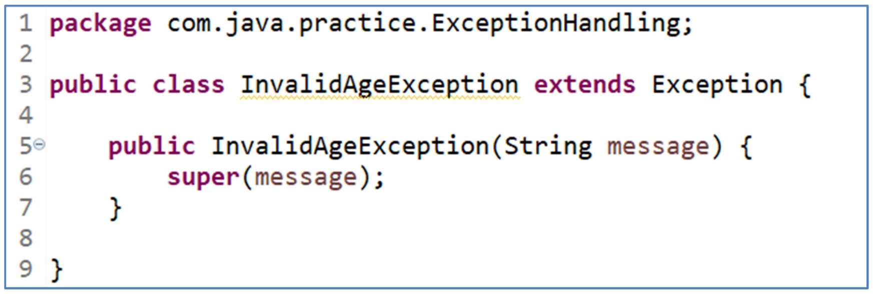
```java
package com.java.practice.ExceptionHandling;

import java.io.File;
import java.io.IOException;

public class ThrowsExample {

    public static void validateFile() throws IOException {
        File file = new File("example.txt");
        if (!file.exists()) {
            throw new IOException("File not found.");
        }
    }

    public static void main(String[] args) {
        try {
            validateFile();
        } catch (IOException e) {
            System.out.println("IOException occurred.");
        }
    }
}
```

Example # 5:

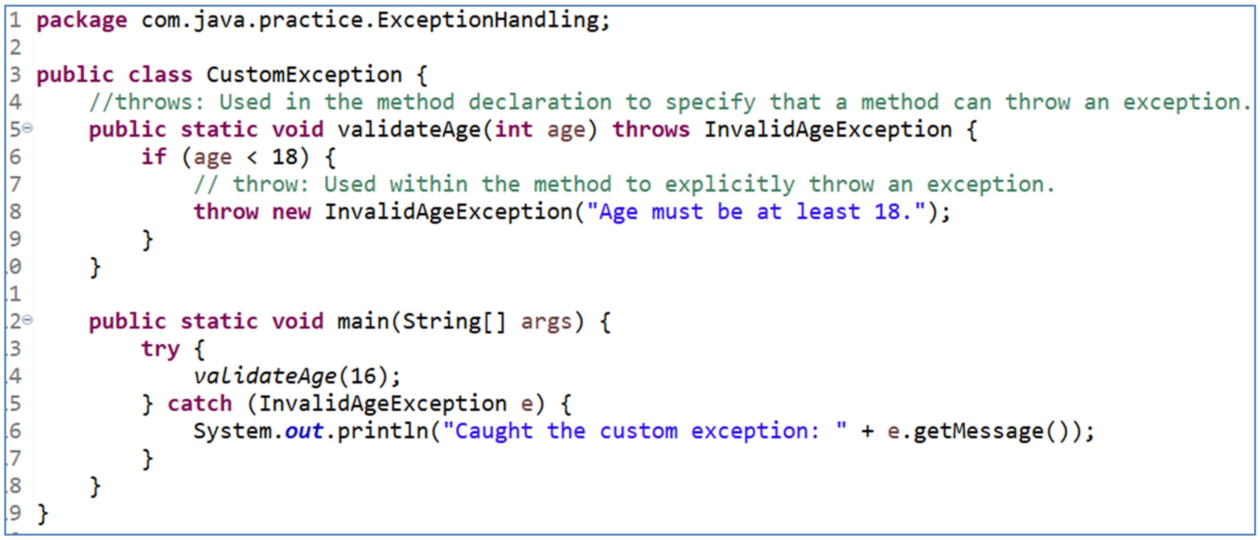


```java
package com.java.practice.ExceptionHandling;

import java.io.FileWriter;
import java.io.IOException;

public class TryWithResourceExample {

    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("example.txt")) {
            writer.write("This file will be closed automatically.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}

```

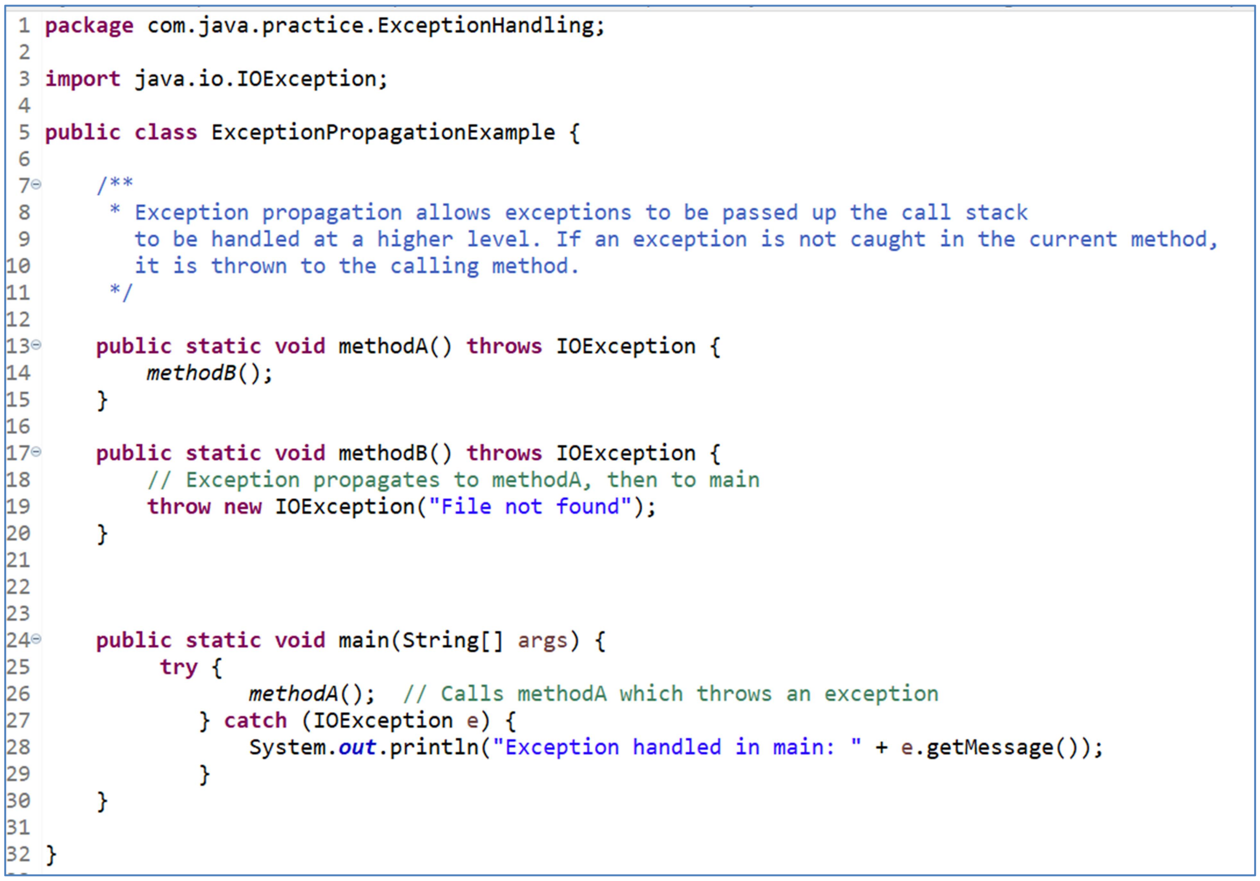
Example # 6:



```java
package com.java.practice.ExceptionHandling;

public class InvalidAgeException extends Exception {

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

}
```



```java
package com.java.practice.ExceptionHandling;

public class CustomException {
    //throws: Used in the method declaration to specify that a method can throw an exception.
    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            // throw: Used within the method to explicitly throw an exception.
            throw new InvalidAgeException("Age must be at least 18.");
        }
    }

    public static void main(String[] args) {
        try {
            validateAge(16);
        } catch (InvalidAgeException e) {
            System.out.println("Caught the custom exception: " + e.getMessage());
        }
    }
}
```



```java
package com.java.practice.ExceptionHandling;

import java.io.IOException;

public class ExceptionPropagationExample {

    /**
     * Exception propagation allows exceptions to be passed up the call stack
       to be handled at a higher level. If an exception is not caught in the current method,
       it is thrown to the calling method.
     */

    public static void methodA() throws IOException {
        methodB();
    }

    public static void methodB() throws IOException {
        // Exception propagates to methodA, then to main
        throw new IOException("File not found");
    }



    public static void main(String[] args) {
         try {
                methodA();  // Calls methodA which throws an exception
            } catch (IOException e) {
                System.out.println("Exception handled in main: " + e.getMessage());
            }
    }

}
```