The *finally* block, used after the try-catch structure, ensures that a particular segment of code runs regardless of whether an exception was thrown in the try block. This is typically used for cleanup operations, like closing resources.

Key points about finally block

- The *finally* block is a block that is used to execute important code such as closing connection, stream, etc.
- Java *finally* block is always executed whether an exception is handled or not.
- Java *finally* block follows the try/catch block. For each try block, there can be zero or more catch blocks, but only one *finally* block.
- The *finally* block will not be executed if the program exits(either by calling *System.exit()* or by causing a fatal error that causes the process to abort).

Syntax:

```
try {
    // Code that might throw an exception
} catch (ExceptionType1 e1) {
    // Code to handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Code to handle ExceptionType2
}
// ... more catch blocks if necessary ...
finally {
    // Code to be executed always, whether an exception occurred or not
}
```

Example 1. Closing a Database Connection using finally Block

Often, when working with databases, it's important to close the connection after performing operations to free up resources and prevent potential connection leaks.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

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

```
        Connection connection = null;

        try {
            connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "user", "password");
            // Perform some database operations
        } catch (SQLException e) {
            System.out.println("Error while accessing the database: " +
e.getMessage());
        } finally {
            try {
                if (connection != null) {
                    connection.close();
                    System.out.println("Database connection closed successfully!");
                }
            } catch (SQLException e) {
                System.out.println("Error while closing the database connection: " +
e.getMessage());
            }
        }
    }
}
```

Example 2. Closing a File Stream using finally Block

When dealing with IO operations, there might be a situation when an exception is thrown, preventing the normal flow of the program. Using *finally* ensures that the stream gets closed, preventing potential resource leaks.

```
import java.io.FileInputStream;
import java.io.IOException;

public class FileStreamExample {
    public static void main(String[] args) {
        FileInputStream fis = null;

        try {
            fis = new FileInputStream("sample.txt");
            int content;
            while ((content = fis.read()) != -1) {
                System.out.print((char) content);
            }
        } catch (IOException e) {
            System.out.println("Error while reading the file: " + e.getMessage());
        } finally {
            try {
                if (fis != null) {
```

```
                    fis.close();
                    System.out.println("\nFile stream closed successfully!");
                }
            } catch (IOException e) {
                System.out.println("Error while closing the file stream: " +
e.getMessage());
            }
        }
    }
}
```

These examples emphasize the significance of the *finally* block in ensuring resource management and cleanup operations are executed reliably.

Scenarios

The *finally* block is designed to execute no matter what, after the try and catch blocks. But there are some scenarios where even the *finally* block might not execute.

1. Using System.exit() in try or catch block

The *System.exit()* method halts the JVM, so if it's called before the *finally* block can execute, then finally won't run.

```
public class FinallyExitExample {

    public static void main(String[] args) {
        try {
            System.out.println("Inside try block");
            System.exit(0); // Terminates JVM
        } catch (Exception e) {
            System.out.println("Inside catch block");
        } finally {
            System.out.println("Inside finally block");
        }
    }
}

// Output:
// Inside try block
```

In the above example, the *finally* block is not executed due to the *System.exit(0)* in the try block.

2. Using the return statement in try or catch block

The *finally* block will still execute even if there's a return statement in the try or catch block.

```
public class FinallyReturnExample {

    public static int getValue() {
        try {
            System.out.println("Inside try block");
            return 1;
        } catch (Exception e) {
            System.out.println("Inside catch block");
            return 2;
        } finally {
            System.out.println("Inside finally block");
        }
    }

    public static void main(String[] args) {
        int result = getValue();
        System.out.println("Returned value: " + result);
    }
}

// Output:
// Inside try block
// Inside finally block
// Returned value: 1
```

In this example, even though the return statement is executed in the try block, the *finally* block still runs before the method returns.