Coding Cheat Sheet: Robust Exception Handling

This reading provides a reference list of code that you'll encounter as you learn exception handling, a critical part of the Java Collections framework. You will learn to handle errors using try-catch blocks, implement custom exceptions, and run code regardless of exceptions using blocks. Understanding these concepts will help you write and debug your first Java programs. Let's explore the following Java coding concepts:

- An Introduction to Exceptions
- Using Finally Block
- Using Multiple Try Catch
- Checked and Runtime Exceptions Compared

Keep this summary reading available as a reference as you progress through your course, and refer to this reading as you begin coding with Java after this course!

An Introduction to Exceptions

Basic exception handling

Description	Example

Java provides a robust mechanism for handling exceptions using the try, catch, and finally blocks.	`public class ExceptionExample {`
	`  public static void main(String[] args) {`
	`  int numerator = 10;`
	`  int denominator = 0; // This will cause an ArithmeticException`
	`  try {`
	`  int result = numerator / denominator; // This line may throw an exception`
	`  System.out.println("Result: " + result);`
	`  } catch (ArithmeticException e) {`
	`  System.out.println("Error: Cannot divide by zero.");`
	`  } finally {`
	`  System.out.println("This block executes regardless of an exception.");`
	`  }`
	`  }`
	`}`

Creating custom exceptions

Description	Example

Sometimes, you may want to create your own exception types. You can do this by extending the Exception class.	`class MyCustomException extends Exception {`
	`  public MyCustomException(String message) {`
	`  super(message); // Pass the message to the parent Exception class`
	`  }`
	`}`
	`public class CustomExceptionExample {`
	`  public static void main(String[] args) {`
	`  try {`
	`  throw new MyCustomException("This is a custom exception message.");`
	`  } catch (MyCustomException e) {`
	`  System.out.println(e.getMessage());`
	`  }`
	`  }`
	`}`

Using Finally Block

The structure of exception-handling

Description	Example

One important feature of Java is its exception handling mechanism, which includes the try, catch, and finally blocks.	`try {`
	`  // Code that may throw an exception`
	`} catch (ExceptionType e) {`
	`  // Code to handle the exception`
	`} finally {`
	`  // Code that will always execute`
	`}`

Description	Example

Understanding the finally block

Description	Example
The code within the finally block always runs after the try and catch blocks, regardless of whether an exception was thrown or caught. It is commonly used for resource management.	`public class FinallyExample { public static void main(String[] args) { try { System.out.println("In try block"); } catch (ArithmeticException e) { int result = 10 / 0; // This line will throw an exception } catch (ArithmeticException e) { System.out.println("Caught an exception: " + e.getMessage()); } finally { System.out.println("Finally block executed"); } } }`

Correct usage of the finally block

Description	Example
Always executing cleanup code: The primary purpose of the finally block is to ensure that the cleanup code runs regardless of whether an exception was thrown or not.	`public class CorrectFinallyUsage { public static void main(String[] args) { FileReader file = null; try { file = new FileReader("example.txt"); // Code to read from the file } catch (IOException e) { System.out.println("Error reading file: " + e.getMessage()); } finally { try { if (file != null) { file.close(); System.out.println("File closed successfully."); } } catch (IOException e) { System.out.println("Error closing file: " + e.getMessage()); } } }`
Releasing database connections: Using the finally block to close database connections is another correct practice.	`import java.sql.Connection; import java.sql.DriverManager; import java.sql.SQLException; public class DatabaseConnectionCorrectUsage { public static void main(String[] args) { Connection connection = null; try { connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "user", "password"); // Perform database operations } catch (SQLException e) { System.out.println("Database error: " + e.getMessage()); } finally { try { if (connection != null) { connection.close(); System.out.println("Database connection closed."); } } catch (SQLException e) { System.out.println("Error closing connection: " + e.getMessage()); }`

Description	Example
	} }

Incorrect usage of the finally block

Description	Example

Not handling exceptions in finally: If an exception occurs in the finally block, it may suppress exceptions thrown in the try or catch blocks.	public class IncorrectFinallyUsage {
	public static void main(String[] args) {
	try {
	int result = 10 / 0; // This will throw an exception
	} catch (ArithmeticException e) {
	System.out.println("Caught an exception: " + e.getMessage());
	} finally {
	int x = 10 / 0; // This will throw another exception
	System.out.println("Finally block executed");
	}
	}
	}

Using return statements in finally: This can lead to unexpected behavior, as it overrides any return statements from the try or catch blocks.	public class ReturnInFinally {
	public static void main(String[] args) {
	System.out.println(testMethod());
	}
	public static int testMethod() {
	try {
	return 1; // Return from try block
	} catch (Exception e) {
	return 2; // Return from catch block
	} finally {
	return 3; // This will override previous returns
	}
	}
	}

Assuming finally will not execute: This is incorrect; the finally block will always execute unless the program crashes or is forcibly terminated.	public class FinallyAlwaysExecutes {
	public static void main(String[] args) {
	try {
	System.out.println("Trying risky operation...");
	int result = 10 / 0; // Throws ArithmeticException
	} catch (ArithmeticException e) {
	System.out.println("Caught an ArithmeticException.");
	// Not exiting or terminating the program here
	}
	// Assuming finally won't execute (this is wrong)
	}
	// The finally block should be here to ensure it executes.
	}

Using Multiple Try Catch

Basic try-catch structure

Description	Example
Multiple try-catch blocks allow developers to handle different types of exceptions in a structured way. In Java, the basic structure of a try-catch block looks like this.	`try { // Code that may throw an exception } catch (ExceptionType e) { // Code to handle the exception }`

Multiple try-catch

Description	Example
Multiple try-catch refers to the use of more than one catch block within a single try statement or multiple try-catch statements in the code.	`public class MultipleCatchExample { public static void main(String[] args) { int[] numbers = {1, 2, 3}; int index = 5; // Invalid index try { // Trying to access an invalid index System.out.println("Number: " + numbers[index]); // Trying to divide by zero int result = 10 / 0; } catch (ArrayIndexOutOfBoundsException e) { System.out.println("Error: Index out of bounds."); } catch (ArithmeticException e) { System.out.println("Error: Division by zero."); } } }`

Throws keyword

Description	Example
The throws keyword is used in method declarations to indicate that a method can throw one or more exceptions.	`public class ThrowsExample { public static void main(String[] args) { try { readFile("nonexistentfile.txt"); } catch (IOException e) { System.out.println("Error: " + e.getMessage()); } } // Method that declares an exception static void readFile(String fileName) throws IOException { FileReader file = new FileReader(fileName); BufferedReader fileInput = new BufferedReader(file); // Reading the file System.out.println(fileInput.readLine()); fileInput.close(); } }`

Checked and Runtime Exceptions Compared

Checked exceptions

Description	Example

Checked exceptions are exceptions checked at compile time. They usually represent recoverable conditions, such as file not found or network issues.	`import java.io.File;`
	`import java.io.FileNotFoundException;`
	`import java.util.Scanner;`
	`public class CheckedExceptionExample {`
	`  public static void main(String[] args) {`
	`  try {`
	`  File myFile = new File("nonexistentfile.txt");`
	`  Scanner myReader = new Scanner(myFile);`
	`  while (myReader.hasNextLine()) {`
	`  String data = myReader.nextLine();`
	`  System.out.println(data);`
	`  }`
	`  myReader.close();`
	`  } catch (FileNotFoundException e) {`
	`  System.out.println("An error occurred: " + e.getMessage());`
	`  }`
	`  }`
	`}`

Runtime exceptions

Description	Example

Runtime exceptions do not need to be explicitly caught or declared. They usually indicate programming errors, such as logic errors or improper use of APIs.	`public class RuntimeExceptionExample {`
	`  public static void main(String[] args) {`
	`  int numerator = 10;`
	`  int denominator = 0;`
	`  try {`
	`  int result = numerator / denominator; // This will cause an ArithmeticException`
	`  System.out.println("Result: " + result);`
	`  } catch (ArithmeticException e) {`
	`  System.out.println("An error occurred: Cannot divide by zero.");`
	`  }`
	`  }`
	`}`

Author(s)

Deepti Adukia
Ramanujam Srinivasan

Other Contributors

Patsy Kravitz