

Week-Session 4- Facilitation Guide **(More about Exception Handling)**

INDEX

- I. Recap from the previous session**
- II. Handling an Entire Class Hierarchy of Exceptions**
- III. Exception Matching**
- IV. Exception Declaration and the Public Interface**
- V. Rethrowing the Same Exception**
- VI. Common Errors**

(2 hrs) ILT

I. Recap from the previous session

In our last session we discussed the Exceptions in Java. We learnt about what Exceptions are and why Exception Handling is required, How to handle and catch the exceptions using try, catch and finally block and how to propagate uncaught exceptions.

In this session we will learn more about exception matching, handling different types of exceptions, rethrowing exceptions and so on.

II. Handling an Entire Class Hierarchy of Exceptions

Handling an entire class hierarchy of exceptions involves using the concept of exception handling to catch and manage exceptions that can be thrown by various methods and operations. Java provides a structured way to handle exceptions using try-catch blocks, and this approach can also be applied to handle a hierarchy of exceptions.

Here's how you can handle an entire class hierarchy of exceptions in Java:

- **Understand the Exception Hierarchy:** Java's exception hierarchy is structured with the base class `java.lang.Exception` at the top and various specific exception classes extending from it. When you want to catch exceptions from a hierarchy, it's important to understand this hierarchy and which exceptions can be caught.
- **Use a Try-Catch Block:** Use a try block to enclose the code that may potentially throw exceptions. Within the try block, you can have multiple catch blocks to handle different types of exceptions.

- **Catch Specific Exceptions First:** Place catch blocks in order from most specific to more general exceptions. This is important because Java will use the first matching catch block for the exception that's thrown.
- **Catch the Base Exception:** If you want to handle a whole hierarchy of exceptions with a common code block, catch the base exception class `java.lang.Exception` at the end of your catch blocks. However, this is generally considered less recommended because it might make your code less maintainable and harder to debug.

An example that demonstrates handling a class hierarchy of exceptions using user input:

```
import java.util.*;
public class ExceptionHandlingWithUserInput
{
    public static void main(String[] args)
    {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter a dividend: ");
            int dividend = scanner.nextInt();

            System.out.print("Enter a divisor: ");
            int divisor = scanner.nextInt();

            int result = divide(dividend, divisor);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught: " + e.getMessage());
        } catch (InputMismatchException e) {
            System.out.println("InputMismatchException caught: Invalid input.");
        } catch (Exception e) {
            System.out.println("General Exception caught: " + e.getMessage());
        } finally {
            scanner.close();
        }
    }
}

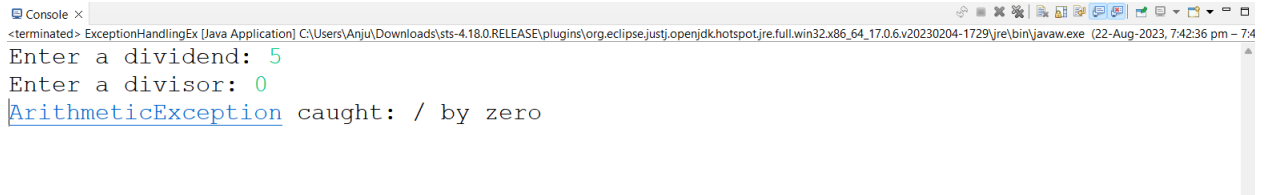
public static int divide(int dividend, int divisor) {
```

```

        return dividend / divisor;
    }
}

```

Output:



```

Console x
<terminated> ExceptionHandlingEx [Java Application] C:\Users\Anju\Downloads\sts-4.18.0.RELEASE\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (22-Aug-2023, 7:42:36 pm - 7:4
Enter a dividend: 5
Enter a divisor: 0
ArithmeticException caught: / by zero

```



```

Console x
<terminated> ExceptionHandlingEx [Java Application] C:\Users\Anju\Downloads\sts-4.18.0.RELEASE\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (22-Aug-2023, 7:43:28 pm - 7:4
Enter a dividend: abc
InputMismatchException caught: Invalid input.

```

In this example, the program takes user input for a dividend and a divisor. It then attempts to divide the dividend by the divisor using the divide method. The try block wraps the input and division operations, and different types of exceptions are caught in separate catch blocks.

1. If the user enters an invalid numeric input, an `InputMismatchException` is caught.
2. If the user attempts to divide by zero, an `ArithmeticException` is caught.
3. If any other exception occurs, a more general `Exception` catch block handles it.

The finally block ensures that the Scanner is closed regardless of whether exceptions are thrown or not.

III. Exception Matching

Exception matching is the process of determining which catch block should handle a thrown exception based on the type of the exception. Java uses the type of the exception to match it to the appropriate catch block. This allows you to handle different types of exceptions in different ways.

When an exception is thrown within a try block, Java will search through the catch blocks associated with that try block in the order they appear. It will match the type of the thrown exception with the types specified in the catch blocks and execute the code inside the first matching catch block.

Here's an example to illustrate exception matching in Java:

```

import java.util.InputMismatchException;
import java.util.Scanner;

```

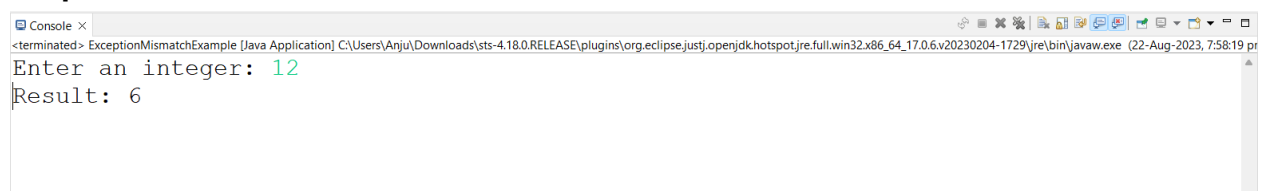
```

public class ExceptionMismatchExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.print("Enter an integer: ");
            int userInput = scanner.nextInt();
            int result = divideBy(userInput, 2);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught: " + e.getMessage());
        } catch (InputMismatchException e) {
            System.out.println("InputMismatchException caught: Invalid input.");
        } catch (Exception e) {
            System.out.println("General Exception caught: " + e.getMessage());
        } finally {
            scanner.close();
        }
    }

    public static int divideBy(int dividend, int divisor) {
        return dividend / divisor;
    }
}

```

Output:



Console ×
 <terminated> ExceptionMismatchExample [Java Application] C:\Users\Anju\Downloads\sts-4.18.0.RELEASE\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (22-Aug-2023, 7:58:19 pm)
 Enter an integer: 12
 Result: 6



Console ×
 <terminated> ExceptionMismatchExample [Java Application] C:\Users\Anju\Downloads\sts-4.18.0.RELEASE\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (22-Aug-2023, 8:00:31 pm)
 Enter an integer: abc
 InputMismatchException caught: Invalid input.

In this example:

The program takes user input for an integer. It attempts to divide the user input by 2 using the divideBy method.

Different types of exceptions are caught in separate catch blocks:

- If the user enters an invalid numeric input (non-integer), an `InputMismatchException` is caught.

- If the user enters a number that would result in division by zero, an `ArithmeticException` is caught.
- If any other exception occurs, a more general `Exception` catch block handles it.

IV. Exception Declaration and the Public Interface

Exception declaration and the public interface are related to how exceptions are communicated to the calling code and how they are specified in method signatures.

Let's explore both concepts:

- **Exception Declaration:** Exception declaration refers to specifying in the method signature the exceptions that a method can potentially throw. This provides information to the caller about the exceptional conditions that the method might encounter. It's part of the method's contract and helps the caller understand what kind of error handling might be required when calling the method.

throw and throws keywords

throw

The `throw` keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exceptions. The `throw` keyword is mainly used to throw custom exceptions.

Example:

// Java program that demonstrates the use of throw

```
class ThrowExcep
{
    static void fun()
    {
        try {
            throw new NullPointerException("demo");
        }
        catch (NullPointerException e) {
            System.out.println("Caught inside fun().");
            throw e; // rethrowing the exception
        }
    }

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

```

        fun();
    }
    catch (NullPointerException e)
    {
        System.out.println("Caught in main.");
    }
}
}

```

Output:

```

Caught inside fun().
Caught in main.

```

throws

The throws is a keyword in Java that is used in the signature of a method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Example:

```

// Java program to demonstrate working of throws
class ThrowsExcep
{
    static void fun() throws IllegalAccessException
    {
        System.out.println("Inside fun(). ");
        throw new IllegalAccessException("demo");
    }

    public static void main(String args[])
    {
        try {
            fun();
        }
        catch (IllegalAccessException e) {
            System.out.println("caught in main.");
        }
    }
}

```

```
}
```

Output:

```
Inside fun().  
caught in main.
```

In Java, you can declare checked exceptions using the `throws` keyword in the method signature. Checked exceptions are those that are checked at compile time, and the calling code is forced to handle or propagate these exceptions.

```
public void readFile(String fileName) throws IOException  
{  
    // Code that might throw an IOException  
}
```

In this example, the `readFile` method is declared to potentially throw an `IOException`. Any calling code that invokes this method needs to handle or declare the `IOException` in a `throws` clause.

Public Interface and Exception Handling: A public interface defines the methods that a class exposes to its clients. When designing the public interface, you need to consider how exceptions are handled and conveyed to the users of your classes.

- **Exception Transparency:** The public interface should make it clear which exceptions can be thrown by the methods. If a method might throw specific exceptions, those exceptions should be documented in the method's documentation or explicitly declared in the method signature.
- **Checked vs. Unchecked Exceptions:** Choose whether to use checked exceptions (forcing the caller to handle or declare them) or unchecked exceptions (like `RuntimeException` subclasses, which the caller isn't required to handle). Checked exceptions are more suitable when the caller can reasonably recover from the exceptional situation, while unchecked exceptions are generally used for unrecoverable conditions or programming errors.
- **Providing Contextual Information:** Exception messages should provide useful information to the caller for diagnosing the problem. This can help the caller understand the root cause of the exception and take appropriate actions.
- **Wrap Exceptions When Necessary:** When your method internally catches an exception but wants to convey information to the caller, you can wrap the caught

exception in a more meaningful exception that's specific to your method's context.

- **Don't Swallow Exceptions:** If you catch exceptions within your methods, make sure not to swallow them without providing appropriate information. Logging or rethrowing a wrapped exception might be necessary to maintain transparency.

V. Rethrowing the Same Exception in java

Rethrowing an exception to catch an exception in one part of your code and then throwing the same exception (or a wrapped version of it) to be handled by higher-level code. This is often done when you want to add additional context or logging to the exception, but you still want the exception to propagate up the call stack for further handling.

Here's an example of how to rethrow the same exception in Java:

```
public class RethrowExceptionExample
{
    public static void main(String[] args)
    {
        try
        {
            performOperation();
        } catch (CustomException e)
        {
            System.out.println("Caught exception in main: " + e.getMessage());
        }
    }
}

public static void performOperation() throws CustomException {
    try {
        // Some operation that might throw an exception
        throw new CustomException("Something went wrong.");
    } catch (CustomException e) {
        System.out.println("Caught exception in performOperation: " + e.getMessage());
        // Rethrow the same exception or a wrapped version of it
        throw e;
    }
}
```

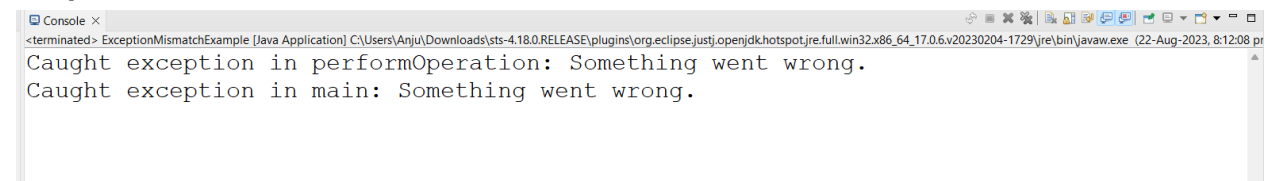


```

class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

```

Output:



```

<terminated> ExceptionMismatchExample [Java Application] C:\Users\Anju\Downloads\sts-4.18.0.RELEASE\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (22-Aug-2023, 8:12:08 pr
Caught exception in performOperation: Something went wrong.
Caught exception in main: Something went wrong.

```

In this example:

The CustomException class is a custom exception class that extends Exception. The performOperation method attempts to perform some operation that might throw a CustomException. Within the try block, if the exception is caught, additional information is printed, and then the same exception is rethrown using throw e;.

VI. Common Errors

Errors are a type of Throwable that typically represent serious, unrecoverable problems that can lead to abnormal program termination. Errors are usually caused by external factors or severe system issues, and they are not meant to be caught and handled by the application code. Instead, they signal critical issues that often require manual intervention or system-level fixes.

Here are some common types of errors in Java:

- **OutOfMemoryError:** This error occurs when the Java Virtual Machine (JVM) cannot allocate enough memory to fulfill an allocation request, usually due to excessive memory usage by the application.
- **StackOverflowError:** This error is thrown when the call stack becomes full, often due to excessive recursion. It indicates that the depth of method calls has exceeded the JVM's stack size limit.
- **NoClassDefFoundError:** This error occurs when the JVM cannot find the definition of a class at runtime, even though the class was available during compilation. This could be due to issues with classpath configuration or missing class files.
- **ClassFormatError:** This error occurs when the JVM encounters an invalid class file format. It might indicate a corrupted or mismatched class file.

- **UnsatisfiedLinkError:** This error is thrown when the JVM cannot find a native library required by the application. It's often related to issues with loading native libraries.
- **InternalError:** This error indicates internal JVM errors that should not normally occur. It might be caused by bugs or inconsistencies within the JVM itself.
- **UnknownError:** This is a subclass of Error that can be used for unexpected errors that don't fit into other error categories.

OutOfMemoryError example in java:

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
public class OutOfMemoryErrorExample
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        List<byte[]> memoryList = new ArrayList<>();
```

```
        try
```

```
        {
```

```
            while (true)
```

```
            {
```

```
                byte[] memoryChunk = new byte[1024 * 1024]; // Allocate 1 MB
```

```
                memoryList.add(memoryChunk);
```

```
            }
```

```
        }
```

```
        catch (OutOfMemoryError e)
```

```
        {
```

```
            System.out.println("OutOfMemoryError caught: " + e.getMessage());
```

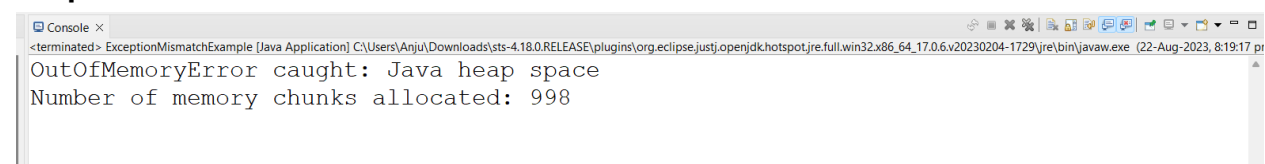
```
            System.out.println("Number of memory chunks allocated: " + memoryList.size());
```

```
        }
```

```
    }
```

```
}
```

Output:



```
Console x
<terminated> ExceptionMismatchExample [Java Application] C:\Users\Anju\Downloads\sts-4.18.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (22-Aug-2023, 8:19:17 pm)
OutOfMemoryError caught: Java heap space
Number of memory chunks allocated: 998
```

In this example, a List is used to hold byte arrays representing memory chunks. The program enters a loop and continuously allocates memory chunks of size 1 MB until the JVM runs out of memory. At that point, an OutOfMemoryError will be thrown.

After the 2 hr LMS, the student has to do a 2 hour live lab. Please refer to the Session 2 Lab doc in the LMS.