Day 5- Facilitation Guide

Index

I. **Exception Handling**

For (1.5 hrs) ILT

In the previous session, we delved into the topics of Polymorphism, Abstraction, and Interfaces. We explored the mechanics of polymorphism, discussed the benefits and drawbacks of abstraction, and highlighted the advantages of using interfaces over abstract classes.

In this session we will explore Exception handling with examples.

Before proceeding further let us summarize some of the important lessons we learned so far:

1.Polymorphism is a fundamental concept in object-oriented programming that allows objects to be treated as instances of their superclass or interface.It enables a single method name to be used for multiple related methods, which can have different implementations based on the context.There are two types of polymorphism in Java: compile-time (static) polymorphism achieved through method overloading and runtime (dynamic) polymorphism achieved through method overriding.

2.An abstract class in Java is a class that cannot be instantiated directly but serves as a blueprint for creating subclasses.It can have both abstract and concrete (non-abstract) methods.Abstract methods are declared without a body and must be implemented by the concrete subclasses.Abstract classes can have instance variables and constructors, just like regular classes.To declare an abstract class, use the abstract keyword before the class declaration.

3.An interface in Java is a reference type that contains abstract methods, default methods, and static methods.It serves as a contract that defines a set of methods that a class implementing the interface must implement.Interfaces provide a way to achieve multiple inheritance in Java, as a class can implement multiple interfaces.All methods in an interface are implicitly public and abstract (before Java 8), and they don't have a method body.Starting from Java 8, interfaces can have default methods, which provide a default implementation for a method that implementing classes can choose to override or use as-is.Use the interface keyword to define an interface in Java.

I. Exception Handling

Exception handling in Java is one of the powerful *mechanisms to handle the runtime errors* so that normal flow of the application can be maintained.

What is exception
Dictionary Meaning: Exception is an abnormal condition.

Exception handling in Java is a mechanism that allows developers to gracefully handle unexpected or exceptional situations that may occur during the execution of a program. These exceptional situations are represented by objects called exceptions. When an exception occurs, the normal flow of the program is disrupted, and the program execution jumps to the nearest suitable exception-handling code.

Why does an exception occur?
There can be several reasons that can cause a program to throw an exception. For example:

Major reasons why an exception Occurs:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

What is exception handling
Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

If an exception occurs, which has not been handled by the programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:
An exception generated by the system is given below

```
package com.anudip.example;
public class ExceptionDemo {
        public static void main(String[] args) {
        int a=10,b=0,c;
```

```
        c=a/b;
        System.out.println("Result:"+ c);
        }
}
```

Here,

ExceptionDemo is the class name

main is the method name

ExceptionDemo.java is the filename

java:7 is the line number

This message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions. We handle such conditions and then print a user friendly warning message to the user, which lets them correct the error as most of the time exceptions occur due to bad data provided by the user.

Exception handling in Java offers several advantages that contribute to writing robust and maintainable code:

Error handling without program termination: Exception handling allows you to catch and deal with errors or exceptional situations without terminating the entire program. This ensures that the program can gracefully handle errors and continue executing the remaining code.

Separation of normal and exceptional code paths: With exception handling, you can separate the normal flow of the program from the exceptional flow. This makes the code easier to read, understand, and maintain, as it keeps the error-handling logic separate from the main business logic.

Cleaner and more maintainable code: By handling exceptions, you can avoid nested if-else blocks to check for errors after each method call. This leads to cleaner and more maintainable code as it reduces the clutter of error-checking code throughout the program.

Propagation of errors to higher levels: In Java, exceptions can be caught at various levels of the call stack. This allows errors to be propagated to higher levels where they can be handled appropriately, providing a more comprehensive and organized approach to error management.

Program flow control: Exception handling allows you to define how your program should react to different error scenarios. You can catch specific exceptions and handle them differently, providing control over program flow in case of errors.

Standardized exception hierarchy: Java has a well-defined hierarchy of exception classes, making it easy to identify and handle specific types of exceptions. This simplifies the error-handling process and promotes code consistency.

Overall, exception handling in Java is a powerful feature that enhances the reliability and robustness of programs by providing a systematic way to deal with unexpected situations, ensuring smoother execution even in the presence of errors.

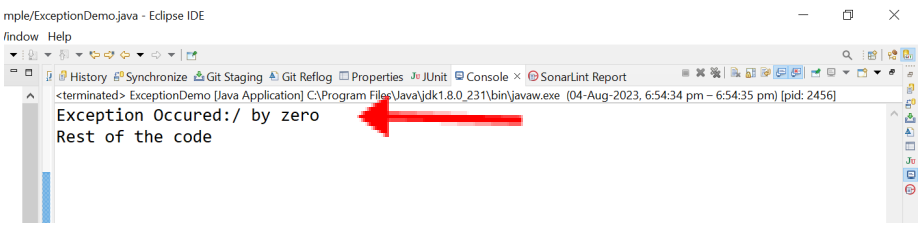
Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If

we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

```
package com.anudip.example;
public class ExceptionDemo {
	public static void main(String[] args) {
	int a=67,b=0;

	try {
		int c=a/b;
		System.out.println(c);
	}
	catch (Exception e) {
		System.out.println("Exception Occured:"+e.getMessage());
	}

	System.out.println("Rest of the code");
	}
}
```



In the above example, you can observe that the program gracefully displays a user-friendly exception message while ensuring the smooth execution of the rest of the code.

Difference between error and exception

In Java, "error" and "exception" are two distinct concepts used to represent different types of problems that can occur during program execution. Here are the key differences between errors and exceptions:

Type of Problem:

Error: Errors represent serious, unrecoverable issues that typically occur at the system level or the virtual machine level. They are often caused by factors outside the control of

the program, such as running out of memory (OutOfMemoryError), stack overflow (StackOverflowError), or JVM-related issues.

Exception: Exceptions, on the other hand, represent exceptional or unexpected situations that occur within the program itself, such as invalid input, resource unavailability, or other runtime errors.

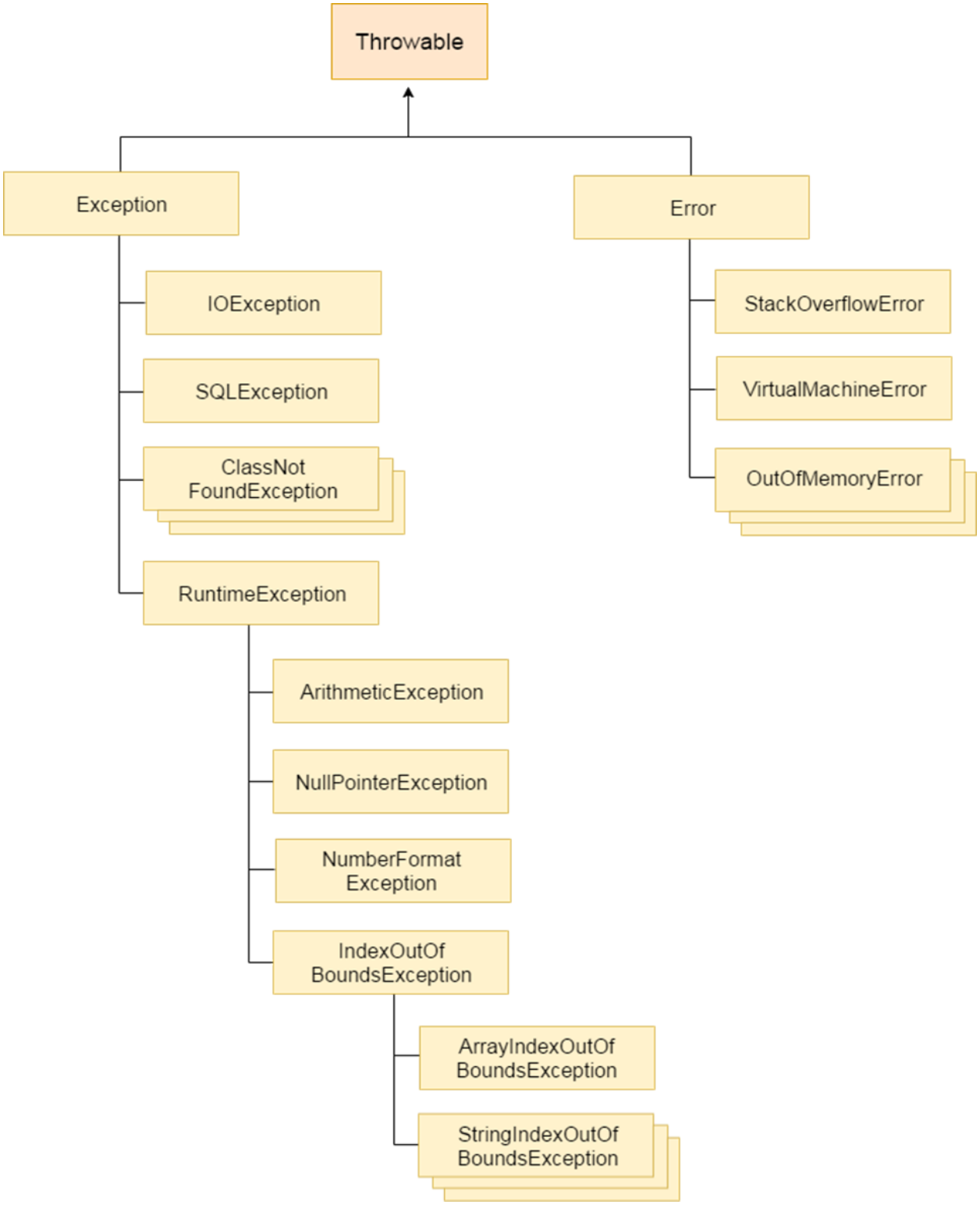
Recoverability:

Error: Errors are generally unrecoverable. Once an error occurs, the program is unlikely to continue its normal execution, and the application may crash or terminate.
Exception: Exceptions are designed to be caught and handled, allowing the program to take appropriate actions and potentially recover from the exceptional situation.

Handling Mechanism:

Error: Errors are not intended to be caught or handled by application code. They are usually not caught by the programmer and are primarily meant for the JVM or system to take appropriate actions.
Exception: Exceptions are explicitly designed for handling by the programmer. Java provides a structured exception-handling mechanism using try-catch blocks to catch and handle exceptions at appropriate places in the code.

Hierarchy of Java Exception classes


Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as an unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions:

1) Checked Exception
The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation errors.

2) Unchecked Exception
The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so the compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit.

3) Error
Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try

2. catch
3. finally
4. throw
5. throws

The try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. In general, a try block looks like the following:

```
try {
   code
}
```

catch and finally blocks . . .

The segment in the example labeled *code* contains one or more legal lines of code that could throw an exception. (The catch and finally blocks are explained in the next two subsections.)

You can put each line of code that might throw an exception within its own try block and provide separate exception handlers for each. Or, you can put all the code within a single try block and associate multiple handlers with it.

```
public class DivisionCalculator {
public static void main(String[] args) {
   Scanner scanner = new Scanner(System.in);

   try {
      System.out.print("Enter the first number: ");
      int num1 = scanner.nextInt();
```

```
            System.out.print("Enter the second number: ");
            int num2 = scanner.nextInt();

            int result = num1/num2;
            System.out.println("Result of division: " + result);
        }

    catch and finally blocks  . . .
}

}
```

If an exception occurs within the try block, that exception is handled by an exception handler associated with it. To associate an exception handler with a try block, you must put a catch block after it.

The catch Block

The catch block is used to catch and handle specific types of exceptions that are thrown within the corresponding try block. If an exception of the specified type occurs in the try block, the corresponding catch block will be executed to handle the exception.

```
try {
} catch (ExceptionType name) {
} catch (ExceptionType name) {
}
```

Each catch block is an exception handler that handles the type of exception indicated by its argument. The argument type, *ExceptionType*, declares the type of

exception that the handler can handle and must be the name of a class that inherits from the Throwable class. The handler can refer to the exception with a name.

The catch block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose *ExceptionType* matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

The following are two exception handlers for the writeList method:

```
try {

} catch (IndexOutOfBoundsException e) {

   System.err.println("IndexOutOfBoundsException: " + e.getMessage());

} catch (IOException e) {

   System.err.println("Caught IOException: " + e.getMessage());

}
```

Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions, as described in the Chained Exceptions section.

Catching More Than One Type of Exception with One Exception Handler:

In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

```
    catch (IOException | SQLException ex) {

   throw ex;}
```

Note: If a catch block handles more than one exception type, then the catch parameter is implicitly final. In this example, the catch parameter *ex* is final and therefore you cannot assign any values to it within the catch block.

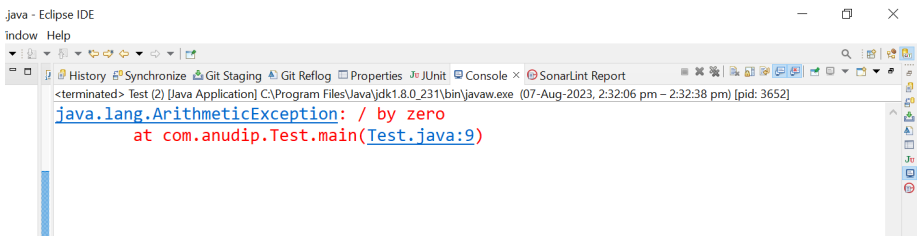
Methods to print the Exception information:

1.printStackTrace()– This method prints exception information in the format of Name of the exception: description of the exception, stack Trace.

//program to print the exception information using printStackTrace() method

```
import java.io.*;

class Test{
   public static void main (String[] args) {
     int a=5;
     int b=0;
      try{
        System.out.println(a/b);
      }
     catch(ArithmeticException e){
      e.printStackTrace();
     }
   }
}
```

Output:

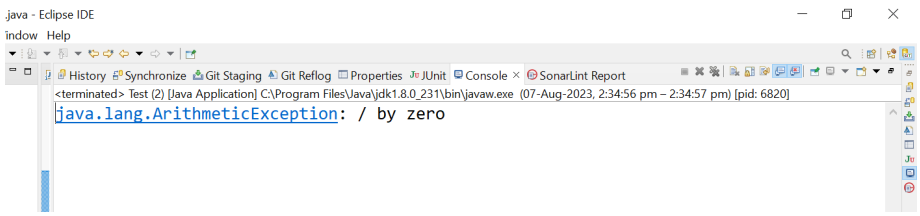


2.toString() – This method prints exception information in the format of Name of the exception: description of the exception.

//program to print the exception information using toString() method

```
import java.io.*;

class Test{
   public static void main (String[] args) {
     int a=5;
     int b=0;
      try{
        System.out.println(a/b);
      }
     catch(ArithmeticException e){
      System.out.println(e.toString());
     }
   }
}
```

Output:



3.getMessage() -This method prints only the description of the exception.

//program to print the exception information using getMessage() method

```
import java.io.*;

class Test{
   public static void main (String[] args) {
     int a=5;
     int b=0;
      try{
        System.out.println(a/b);
      }
     catch(ArithmeticException e){
      System.out.println(e.getMessage());
```

```
    }
  }
}
```

Output:

The finally Block

> The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

Note: The finally block may not execute if the JVM exits while the try or catch code is being executed.

Example:

```
package com.anudip;

public class DivisionExample {
public static void main(String[] args) {
 int num1 = 10;
int num2 = 0;
try {

int result = divideNumbers(num1, num2);
System.out.println("Result of division: " + result);
}
 catch (ArithmeticException e) {
System.out.println("Error: Division by zero is not allowed.");
}
finally {
System.out.println("Finally block executed.");
}
```

```
    }

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

Output:

In this example, the finally block will be executed regardless of whether an exception occurs during the division (in case of division by zero) or not.

Important: *Use a try-with-resources statement instead of a finally block when closing a file or otherwise recovering resources.*

The try-with-resources Statement

The try-with-resources statement is a try statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

The following example uses a try-with-resources statement to clean up and close the PrintWriter and FileWriter for the writeList method:

```
public void writeList() throws IOException {

    try (FileWriter f = new FileWriter("OutFile.txt");  // try-with-resources

        PrintWriter out = new PrintWriter(f)) {

       for (int i = 0; i < SIZE; i++) {

          out.println("Value at: " + i + " = " + list.get(i));
```

```
    }
  }
}
```

In the provided code snippet, the try-with-resources statement is used to handle resources that need to be closed after their operations are completed, even in the presence of exceptions. This feature was introduced in Java 7 and makes resource management more concise and less error-prone.

The try-with-resources statement is used with classes that implement the AutoCloseable interface or its subinterface java.io.Closeable. When the try block is exited, either normally or exceptionally (due to an exception), the resources opened within the parentheses of the try statement will be closed automatically.

Let's break down the code and understand how the try-with-resources statement works:

Resource Declaration:

- The try-with-resources statement is initiated with a set of resource declarations inside the parentheses following the try keyword.
- In this example, two resources are declared within the parentheses: FileWriter and PrintWriter.
- The FileWriter is used to write character data to a file, and the PrintWriter is used to print formatted representations of objects to a text-output stream.

Resource Initialization and Use:

- Within the try block, the resources are initialized and used just like any other code block.
- In this case, the FileWriter is used to open a file named "OutFile.txt" for writing, and the PrintWriter is initialized to wrap the FileWriter.

Resource Auto-Close:

- As soon as the execution of the try block completes, either successfully or due to an exception, the resources are automatically closed in the reverse order of their declaration (from last to first).
- The closing is done by invoking the close() method on the resources.
- The close() method is defined in the AutoCloseable interface, which is implemented by classes like FileWriter and PrintWriter.

Exception Handling:

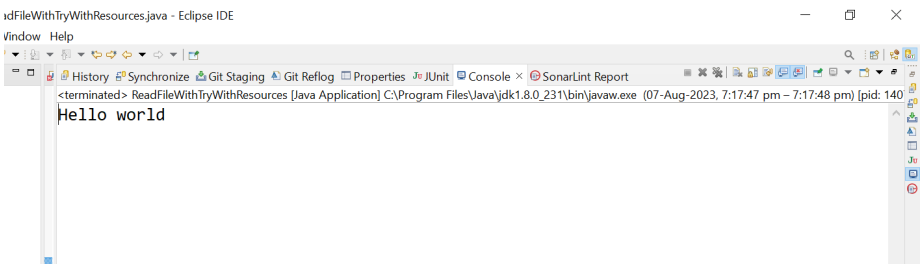
- If an exception occurs during the execution of the try block, the catch block (if present) will be executed, and then the resources will be closed automatically.
- If no exception occurs, the resources are still closed automatically when the execution of the try block completes.

In summary, the try-with-resources statement in this example ensures that the FileWriter and PrintWriter resources are correctly closed after the block's execution, even if an exception occurs during the process. This approach simplifies resource management, reduces boilerplate code, and guarantees proper cleanup of resources, making the code more concise and robust.

Example:

```
package com.anudip;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class ReadFileWithTryWithResources {
	public static void main(String[] args) {
		try (BufferedReader reader =
new BufferedReader(new FileReader("D:\\Anudip Foundation\\D drive all
docs\\new\\New\\input.txt"))) {
		String line;
		while ((line = reader.readLine()) != null) {
		System.out.println(line);
		}
		} catch (IOException e) {
		System.out.println("Error reading the file: " + e.getMessage());
		}
		}
}
```

Output:



Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or unchecked exception in Java by throw keyword. The throw keyword is mainly used to throw custom exception.

The syntax of the java throw keyword is given below.

1. **throw** exception;

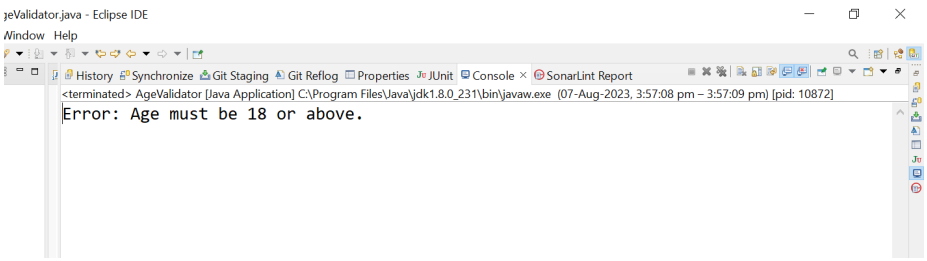
Let's see the example of throwing an IOException.

1. **throw new** IOException("sorry device error");

```
 package com.anudip;
public class AgeValidator {

public void validateAge(int age) throws InvalidAgeException
{
if (age < 18) {
throw new InvalidAgeException("Age must be 18 or above.");
}
// If age is valid, the method continues execution normally.
}
public static void main(String[] args)
        {
                AgeValidator validator=new AgeValidator();
                 try {
         int age = 15;
         validator.validateAge(age);
         System.out.println("Age is valid!");
         } catch (InvalidAgeException e) {
```

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

Output:



Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is the programmer's fault that he is not performing check up before the code being used.

Syntax of java throws

```
return_type method_name() throws exception_class_name{
 //method code
 }
```

Question: ***Which exception needs to be specified or declared using throws clause?***

Advantage of Java throws keyword

Now Checked Exceptions can be propagated (forwarded in call stack).

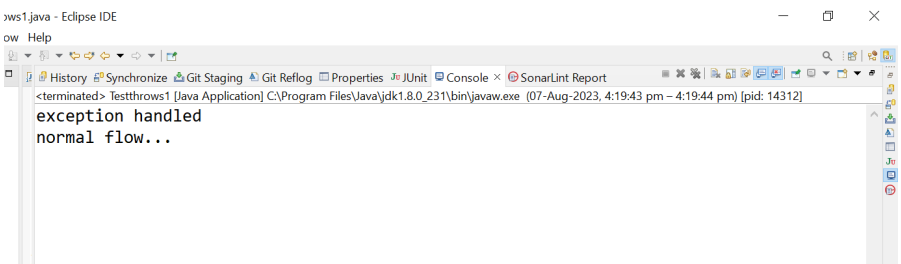
It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of a java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:



The throws clause in Java exceptions encompasses two scenarios:

Case1: If you effectively manage the exception through the use of a try/catch construct, the code will execute smoothly..

Case2: If you have declared the exception by specifying "throws" with the method, but not handled by try/catch, then the exception remains unhandled and will be thrown at runtime and the program will terminate.

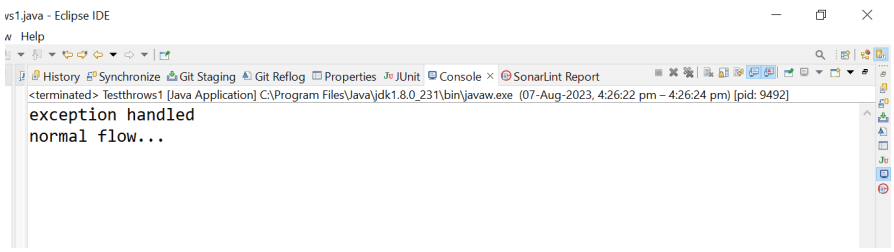
Let's see examples for both scenarios:

Case1: You handle the exception

- If you handle the exception, the code will execute smoothly regardless of whether the exception occurs during the program's execution or not.

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");
  }
}
```

Output:



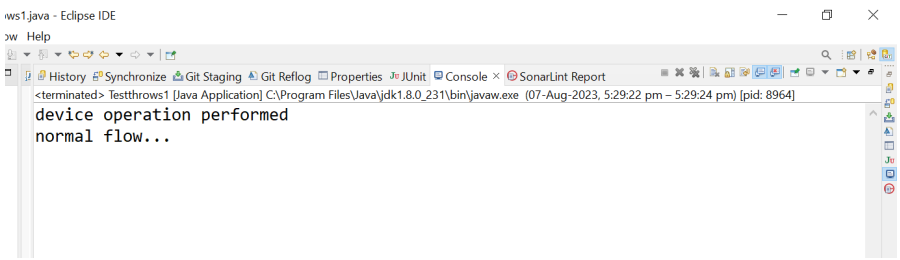
Case2: You declare the exception only using throws clause but not handled by try/catch:

- A) If you specify the exception using throws clause and if it does not occur, the code will execute smoothly.
- B) If you specify the exception using the "throws" clause, and an exception occurs, it will be thrown at runtime because "throws" does not handle the exception.

Note:The purpose of the throws clause is to declare the exception, but it does not take care of handling the exception.

A)Program if exception does not occur

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```



B)*Program if exception does occur,throws clause does not take care of handling the exception.*

```
package com.anudip;
import java.io.IOException;
class M{
        void method()throws IOException{
                throw new IOException("exception occurred");
                }
        }

public class Testthrows1 {
        public static void main(String args[]) throws IOException{
                M m=new M();
                m.method();

                System.out.println("normal flow...");
}
        }
```

Output:

Overall,The throws clause in Java is used to declare the types of exceptions that a method might throw during its execution. It provides information to the caller of the method about the potential exceptions that need to be handled. However, the throws clause itself does not handle the exceptions – it's the responsibility of the calling code to either handle these exceptions using try/catch blocks or propagate them further using additional throws clauses. This mechanism allows for a clear separation between code that generates exceptions and code that deals with them, promoting better code organization and error management.

Custom Exceptions

In larger applications, most of the cases we need custom exceptions for representing business exceptions which are, at a level higher than technical exceptions defined by JDK.

Below are the examples for custom exceptions:

- InvalidAgeException
- ResourceNotFoundException
- BadRequestException
- UnauthorizedRequestException etc.

Writing your own exception class

Here are the steps create a custom exception:

- Create a new class whose name should end with an Exception like *ClassNameException*. This is a convention to differentiate an exception class from regular ones.
- Make the class *extends* one of the exceptions which are subtypes of the *java.lang.Exception* class. Generally, a custom exception class always extends directly from the *Exception* class.
- Create a constructor with a *String* parameter which is the detailed message of the exception. In this constructor, simply call the super constructor and pass the message. In Java, there are two types of exceptions – checked and unchecked exceptions.

Custom Checked Exception

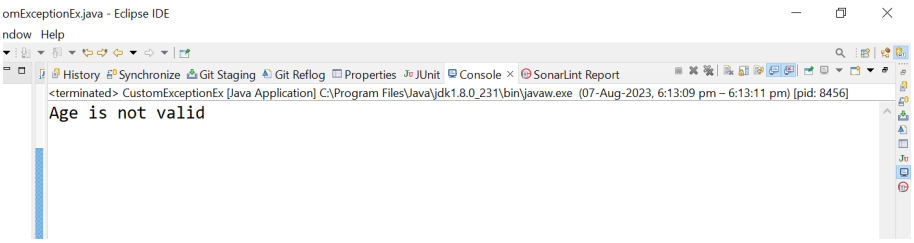
Checked exceptions are exceptions that need to be treated explicitly.

To create a custom exception, we have to extend the *java.lang.Exception* class.

```
// A Class that represents use-defined exception

class AgeValidation extends Exception {
    public AgeValidation (String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class CustomExceptionEx{
    // Driver Program

   static  Void checkAge(int age) throws AgeValidation
     {
           if(age<18)
      {
         throw new AgeValidation ("Age is not valid");

        }
        else
          {
            System.out.println(“age is valid”);
            }

        }
    public static void main(String args[])
    {
        try {
         checkAge(17);
                }
        catch (AgeValidation  ex) {
            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
```

}

Output


omExceptionEx.java - Eclipse IDE
ndow Help
History Synchronize Git Staging Git Reflog Properties JUnit Console SonarLint Report
<terminated> CustomExceptionEx [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (07-Aug-2023, 6:13:09 pm – 6:13:11 pm) [pid: 8456]
Age is not valid