Week 2
Exception Handling

1. What is an Exception?

An *exception* is an error event that can happen during the execution of a program and disrupts its normal flow.

Exception Handling:

Java provides a robust and object-oriented way to handle exception scenarios known as Java Exception Handling.

Example Situation:

Exceptions in Java can arise from different kinds of situations such as wrong data entered by the user, hardware failure, network connection failure, or a database server that is down. The code that specifies what to do in specific exception scenarios is called exception handling.

Working: (throwing)

If an exception occurs in a method, the process of creating the exception object and handing it over to the runtime environment is called *"throwing the exception"*.

Try-block:

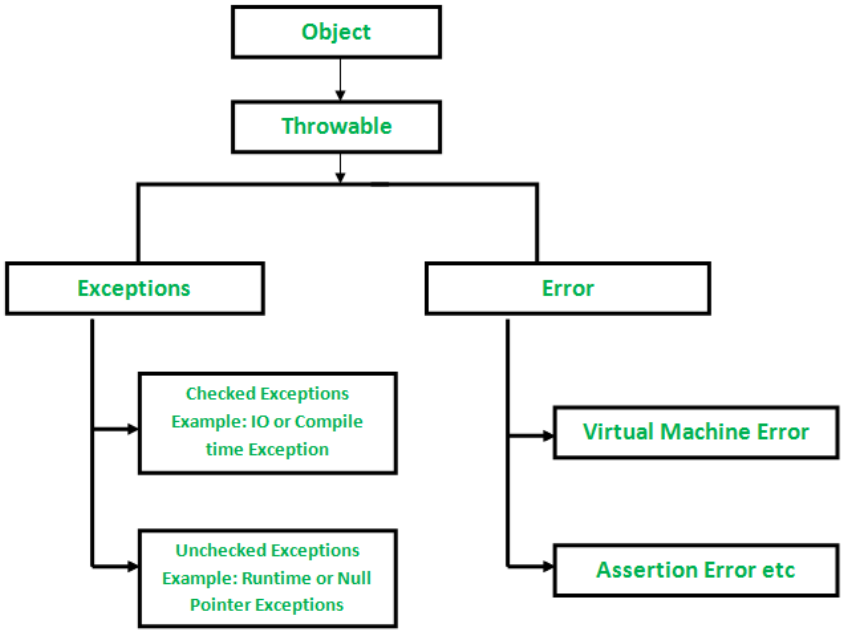
Exception Handler is the block of code that can process the exception object.

- The logic to find the exception handler begins with searching in the method where the error occurred.
- If no appropriate handler is found, then it will move to the caller method.

If an appropriate exception handler is found, the exception object is passed to the handler to process it. The handler is said to be *"catching the exception"*. If there is no appropriate exception handler, found then the program terminates and prints information about the exception to the console.

Java Exception handling framework is used to handle runtime errors only. The compile-time errors have to be fixed by the developer writing the code else the program won't execute.

Hierarchy:



Both Errors and Exceptions are the subclasses of java.lang.Throwable class

Errors v/s Exceptions:

Errors and Exceptions are **both** types of throwable objects, but they represent different types of problems that can occur during the execution of a program.

Errors are usually caused by serious problems that are outside the control of the program, such as running out of memory or a system crash. Errors are represented by the Error class and its subclasses. **Eg:** OutOfMemory, StackOverFlowError, NoClassDefFoundError etc.

Exceptions, on the other hand, are used to handle errors that can be recovered from within the program. Exceptions are represented by the Exception class and its subclasses.

Exceptions can be caught and handled within a program, it's common to include code to catch and handle exceptions in Java programs. By handling exceptions, you can provide more informative error messages to users and prevent the program from crashing.
Eg: NullPointerException, IllegalArgumentException, IOException

In summary, errors and exceptions represent different types of problems that can occur during program execution. Errors are usually caused by serious problems that cannot be recovered from, while exceptions are used to handle recoverable errors within a program.

Types of Exceptions:
1.Checked (Compile Time)
2.Unchecked (Run Time)
Programming errors often remain undetected until the program is compiled or executed. Some of the errors inhibit the program from being compiled or executed. Thus, errors should be removed before compiling and executing.
It is of **three types**: Compile-time, Run-time and Logical

In Java Exceptions refer to an unwanted or unexpected event, which occurs during the execution of a program i.e. at run time, that disrupts the normal flow of the program's instructions.

Exceptions are the conditions that occur at runtime and may cause the termination of the program. But they are recoverable using try, catch and throw keywords. Exceptions are divided into two categories:

- Checked exceptions
- Unchecked exceptions

Checked exceptions like IOException known to the compiler at compile time while **unchecked exceptions** like ArrayIndexOutOfBoundException known to the compiler at runtime. It is mostly caused by the program written by the programmer.

Errors	Exceptions
Recovering from Error is not possible.	We can recover from exceptions by either using try-catch block or throwing exceptions back to the caller.
All errors in java are unchecked type.	Exceptions include both checked as well as unchecked type.
Errors are mostly caused by the environment in which the program is running.	The program itself is responsible for causing exceptions.

Errors can occur at compilation time.	Unchecked exceptions occur at runtime whereas checked exceptions occur at compile time
They are defined in java.lang.Error package.	They are defined in java.lang.Exception package

Java Exception Handling Keywords

Java provides a structured way to handle exceptions using try, catch, and finally blocks.

Let's break down each component:

- **try block**: This is where you place the code that may throw an exception.
- **catch block**: Used to handle specific exceptions that may arise in the try block.
- **finally block**: This block always executes, regardless of whether an exception occurs or not. It's typically used for cleanup tasks like closing resources.

```
try {
   // Risky code
} catch (Exception e) {
   // Handling code
} finally {
   // Clean-up code
}
```

If an exception occurs anywhere within the try block, subsequent statements within the try block will not be executed, even if the exception is handled. Therefore, it's crucial to include only risky code within the try block, and the try block's length should be kept as short as possible.

Apart from the try block, exceptions might also occur within catch and finally blocks. If any statement outside of the try block raises an exception, it always results in abnormal termination.

Multiple Catch blocks:

Each catch block is dedicated to handling a specific type of exception. This approach allows for more precise and targeted handling, improving the robustness and reliability of the code. Additionally, it provides flexibility in dealing with different types of exceptions appropriately.

In a try-with-multiple-catch-blocks scenario, it's crucial to order the catch blocks properly. Child exceptions should be caught before parent exceptions. Failing to do so results in a compile-time error indicating that the exception has already been caught.

Combinations and Rules for Try-Catch-Finally:

- In try-catch-finally, the order is important.
- Whenever we write try, it's compulsory to include either catch or finally; otherwise, we will get a compile-time error (try without catch or finally is invalid).
- Whenever we write a catch block, a try block must be present; catch without try is invalid.
- Whenever we write a finally block, a try block should be present; finally without try is invalid.
- Inside try-catch-finally blocks, we can nest additional try-catch-finally blocks; nesting of try-catch-finally is allowed.
- Curly braces ({}) are mandatory for try-catch-finally blocks.

Nested-try-catch block valid syntax:

```
try {
 try {
 // code
 } catch(Exception e) {
 // inner catch block
 } finally {
 // inner finally block
 }
```

```
} catch(Exception e) {
 // outer catch block
} finally {
 // outer finally block
}
```

In Java, exception handling is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. It handles runtime errors such as NullPointerException, ArrayIndexOutOfBoundsException, etc. To handle these errors effectively, Java provides two keywords, throw and throws.

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.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing try block is checked and so on. If no matching catch is found then the default exception handler will halt the program

Throws

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.

Syntax:
type method_name(parameters) throws ***exception_list***

where, ***exception_list*** is a comma separated list of all the exceptions which a method might throw. We can use the throws keyword to delegate the responsibility of

exception handling to the caller (It may be a method or JVM) then the caller method is responsible to handle that exception.

Difference Between throw and throws

throw	throws
It is used to explicitly throw an exception.	It is used to declare that a method might throw one or more exceptions.
It is used inside a method or a block of code. It can throw both checked and unchecked exceptions.	It is used in the method signature. It is only used for checked exceptions. Unchecked exceptions do not require **throws**
The method or block throws the exception.	The method's caller is responsible for handling the exception.
Stops the current flow of execution immediately.	It forces the caller to handle the declared exceptions.
throw new ArithmeticException("Error");	public void myMethod() throws IOException {}

An exception handler handles a specific class can also handles its subclasses. That's why we put the topmost Exception at the end catch block.

What is `NotImplementedException`?

A `NotImplementedException` is an exception that is thrown to indicate that a particular method or a part of the code has not been fully implemented yet. It serves as a clear signal to other developers (or even to yourself in the future) that the functionality is still under development or has been left as a placeholder for later implementation.

To create a `NotImplementedException` in Java, you need to define a custom exception class that extends the `RuntimeException` class.

```
public class NotImplementedException extends RuntimeException {
    public NotImplementedException() {
        super("This method is not yet implemented.");
    }

    public NotImplementedException(String message) {
        super(message);
    }
}
```

We have created a custom exception class `NotImplementedException` that extends `RuntimeException`. It has two constructors: one with a default error message and another that allows you to provide a custom error message.

The `NotImplementedException` in Java, although not a built - in feature, is a valuable tool for developers. It helps in signaling incomplete functionality, avoiding confusion, and facilitating the development process.

Exception Handling with Method Overriding in Java

Exception handling with method overriding in Java refers to the rules and behavior that apply when a subclass overrides a method from its superclass and both methods involve exceptions.

Rules for Exception Handling with Method Overriding

- The subclass can throw the same or smaller exceptions as the superclass methods.
- The subclass can choose not to throw any exceptions.
- The subclass cannot throw new checked exceptions not in the parent method.

Problem 1:

If Superclass doesn't declare any exception and subclass declare checked exception. A subclass cannot introduce a new checked exception if the superclass method does not declare any.

If Superclass doesn't declare any exception and Subclass declare unchecked exception. ArithmeticException is unchecked, so compiler allows it.

Problem 2:

If the Superclass declares an exception. In this problem 3 cases will arise as follows:

1. If Superclass declares an exception and Subclass declares exceptions other than the child exception of the Superclass declared exception --> subclass can not override method() super calss.