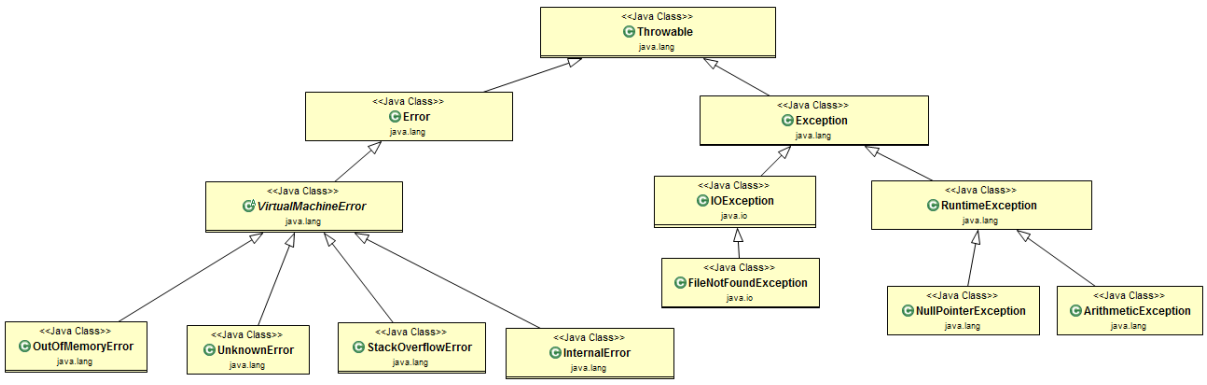
## Introduction

Exception handling in Java is a crucial mechanism that allows developers to manage runtime errors gracefully, ensuring the program continues to run or terminates smoothly. We will see basics of exception handling in Java, covering key concepts, keywords, and examples to help you understand how to handle exceptions effectively.




### 1. What are Exceptions?

Exceptions are events that occur during the execution of a program and disrupt its normal flow. They are objects that represent an error or unexpected behavior. Exception handling provides a way to deal with these events, ensuring the program can recover or terminate gracefully.

### 2. Types of Exceptions

#### Checked Exceptions

Checked exceptions are exceptions that are checked at compile-time. These exceptions must be either caught or declared in the method signature using the throws keyword. They represent conditions that a reasonable application might want to catch.

**Examples:**

- IOException
- SQLException
- FileNotFoundException

#### Unchecked Exceptions

Unchecked exceptions are exceptions that are not checked at compile-time. They are subclasses of RuntimeException. Unchecked exceptions represent programming errors, such as logic mistakes or improper use of an API.

**Examples:**

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException

**Errors**

Errors are serious issues that a reasonable application should not try to catch. They are typically conditions that a program cannot recover from and are external to the application.

**Examples:**

- OutOfMemoryError
- StackOverflowError
- VirtualMachineError

**3. Exception Handling Keywords**

**try**

The try block contains code that might throw an exception. If an exception occurs, it is thrown to the corresponding catch block.

**Syntax:**

```
try {
    // Code that might throw an exception
}
```

**catch**

The catch block handles the exception thrown by the try block. Multiple catch blocks can be used to handle different types of exceptions.

**Syntax:**

```
try {
    // Code that might throw an exception
```

```
} catch (ExceptionType e) {
    // Code to handle the exception
}
```

**finally**

The finally block contains code that will always execute, regardless of whether an exception was thrown or caught. It is typically used for resource cleanup.

**Syntax:**

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
    // Code that will always execute
}
```

**throw**

The throw keyword is used to explicitly throw an exception.

**Syntax:**

```
throw new ExceptionType("Exception message");
```

**throws**

The throws keyword is used in a method signature to declare that the method might throw one or more exceptions.

**Syntax:**

```
returnType methodName(parameterList) throws ExceptionType1, ExceptionType2 {
    // Method body
}
```

**4. Exception Hierarchy**

The exception hierarchy in Java is as follows:

```
java.lang.Object
   └── java.lang.Throwable
       ├── java.lang.Exception
       │    ├── java.io.IOException
       │    ├── java.sql.SQLException
       │    └── java.lang.RuntimeException
       │         ├── java.lang.NullPointerException
       │         ├── java.lang.ArrayIndexOutOfBoundsException
       │         └── java.lang.ArithmeticException
       └── java.lang.Error
            ├── java.lang.OutOfMemoryError
            ├── java.lang.StackOverflowError
            └── java.lang.VirtualMachineError
```

**5. Basic Exception Handling Example**

Let's start with a basic example to understand how to handle exceptions in Java.

**Example:**

```
public class BasicExceptionHandling {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // This will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Caught exception: " + e.getMessage());
        } finally {
            System.out.println("Finally block executed.");
        }
    }
}
```

**Output:**

Caught exception: / by zero

Finally block executed.

**Explanation:**

- The try block contains code that might throw an ArithmeticException.
- The catch block handles the ArithmeticException.
- The finally block is executed regardless of whether an exception was thrown.

**6. Handling Multiple Exceptions**

A method can have multiple catch blocks to handle different types of exceptions separately.

**Example:**

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[10]); // This will throw ArrayIndexOutOfBoundsException
            int result = 10 / 0; // This will throw ArithmeticException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds: " + e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic error: " + e.getMessage());
        }
    }
}
```

**Output:**

Array index out of bounds: Index 10 out of bounds for length 3

**Explanation:**

- The first catch block handles ArrayIndexOutOfBoundsException.
- The second catch block handles ArithmeticException.
- Only the first exception that occurs (ArrayIndexOutOfBoundsException) is caught and handled.

**7. Nested try Blocks**

You can nest try blocks inside each other to handle exceptions that might occur within multiple levels of operations.

**Example:**

```
public class NestedTryExample {
    public static void main(String[] args) {
        try {
            System.out.println("Outer try block");
            try {
                int result = 10 / 0; // This will throw ArithmeticException
            } catch (ArithmeticException e) {
                System.out.println("Inner catch: Arithmetic error: " + e.getMessage());
            }
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[10]); // This will throw ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Outer catch: Array index out of bounds: " + e.getMessage());
        } finally {
            System.out.println("Outer finally block");
        }
```

```
    }
}
```

**Output:**

```
Outer try block
Inner catch: Arithmetic error: / by zero
Outer catch: Array index out of bounds: Index 10 out of bounds for length 3
Outer finally block
```

**8. Custom Exceptions**

You can create your own custom exceptions by extending the Exception class or any of its subclasses. Custom exceptions are useful for specific error conditions that are relevant to your application.

**Example:**

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

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
        }
    }
```

```
    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older.");
        }
        System.out.println("Age is valid.");
    }
}
```

**Output:**

Caught custom exception: Age must be 18 or older.

**Explanation:**

- The InvalidAgeException class extends the Exception class.
- The validateAge method throws an InvalidAgeException if the age is less than 18.
- The exception is caught in the main method.

**9. Chained Exceptions**

Chained exceptions allow you to relate one exception with another, forming a chain of exceptions. This is useful when an exception occurs as a direct result of another exception.

**Example:**

```
public class ChainedExceptionDemo {
    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    public static void method1() throws Exception {
        try {
            method2();
        } catch (Exception e) {
            throw new Exception("Exception in method1", e);
        }
    }

    public static void method2() throws Exception {
        throw new Exception("Exception in method2");
    }
}
```

**Output:**

```
java.lang.Exception: Exception in method1
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:12)
    at ChainedExceptionDemo.main(ChainedException

Demo.java:5)
Caused by: java.lang.Exception: Exception in method2
    at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:17)
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:10)
    ... 1 more
```

**Explanation:**

- method2 throws an exception.
- method1 catches the exception thrown by method2 and throws a new exception with the original exception as the cause.

- The main method catches the exception thrown by method1 and prints the stack trace, showing the chain of exceptions.

**Using finally for Resource Cleanup**

The finally block is typically used for resource cleanup, such as closing files or releasing network resources.

**Example:**

```
import java.io.FileWriter;
import java.io.IOException;


public class FinallyExample {
    public static void main(String[] args) {
        FileWriter writer = null;
        try {
            writer = new FileWriter("example.txt");
            writer.write("Hello, World!");
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        } finally {
            try {
                if (writer != null) {
                    writer.close();
                }
            } catch (IOException e) {
                System.out.println("Failed to close the writer: " + e.getMessage());
```

```
            }
            System.out.println("Finally block executed.");
        }
    }
}
```

**Output:**

Finally block executed.

**Complete Flow Example Program**

Here is a complete program that demonstrates the use of all the exception handling keywords in Java, including handling multiple exceptions and using nested try blocks, showcasing the complete flow of exception handling.

**Example Code:**

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

// Custom exception
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CompleteExceptionHandlingExample {
```

```
    public static void main(String[] args) {
        // Example 1: Using try-catch-finally
        try {
            int result = 10 / 0; // This will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Caught ArithmeticException: " +
e.getMessage());
        } finally {
            System.out.println("Finally block executed.");
        }

        // Example 2: Using throw with a built-in exception
        try {
            validateAge(15);
        } catch (IllegalArgumentException e) {
            System.out.println("Caught IllegalArgumentException: " +
e.getMessage());
        }

        // Example 3: Using throws with a built-in exception
        try {
            readFile("example.txt");
        } catch (FileNotFoundException e) {
            System.out.println("Caught FileNotFoundException: " +
e.getMessage());
        }

        // Example 4: Using throw with a custom exception
```

```
        try {
            validateCustomAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
        }

        // Example 5: Multiple catch blocks
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[10]); // This will throw ArrayIndexOutOfBoundsException
            int result = 10 / 0; // This will throw ArithmeticException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds: " + e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic error: " + e.getMessage());
        }

        // Example 6: Nested try blocks
        try {
            System.out.println("Outer try block");
            try {
                int result = 10 / 0; // This will throw ArithmeticException
            } catch (ArithmeticException e) {
                System.out.println("Inner catch: Arithmetic error: " + e.getMessage());
            }
```

```
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[10]); // This will throw ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Outer catch: Array index out of bounds: " + e.getMessage());
        } finally {
            System.out.println("Outer finally block");
        }

        // Example 7: Demonstrating complete flow
        try {
            System.out.println("Starting the complete flow example.");
            processFile("example.txt");
            validateAgeForDriving(16);
        } catch (FileNotFoundException | InvalidAgeException e) {
            System.out.println("Exception caught in main: " + e.getMessage());
        } finally {
            System.out.println("Cleanup in finally block.");
        }
        System.out.println("End of the complete flow example.");
    }

    public static void validateAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or older.");
        }
```

```
        System.out.println("Age is valid.");
    }

    public static void readFile(String fileName) throws FileNotFoundException {
        File file = new File(fileName);
        FileReader fr = new FileReader(file);
        System.out.println("File read successfully");
    }

    public static void validateCustomAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older.");
        }
        System.out.println("Age is valid.");
    }

    public static void processFile(String fileName) throws FileNotFoundException {
        File file = new File(fileName);
        FileReader fr = new FileReader(file);
        System.out.println("Processing file: " + fileName);
    }

    public static void validateAgeForDriving(int age) throws InvalidAgeException {
        if (age < 18) {
```

```
            throw new InvalidAgeException("Age must be 18 or older to drive.");
        }
        System.out.println("Eligible for driving.");
    }
}
```

**Output:**

Caught ArithmeticException: / by zero

Finally block executed.

Caught IllegalArgumentException: Age must be 18 or older.

Caught FileNotFoundException: example.txt (No such file or directory)

Caught custom exception: Age must be 18 or older.

Array index out of bounds: Index 10 out of bounds for length 3

Outer try block

Inner catch: Arithmetic error: / by zero

Outer catch: Array index out of bounds: Index 10 out of bounds for length 3

Outer finally block

Starting the complete flow example.

Exception caught in main: example.txt (No such file or directory)

Cleanup in finally block.

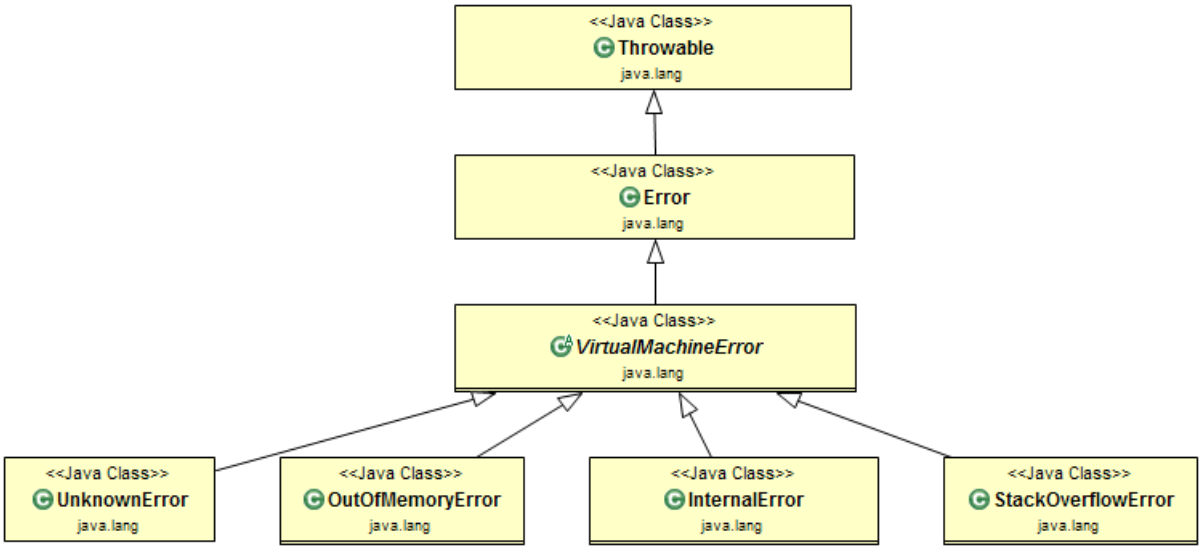
End of the complete flow example.

### 1. Error Class

The *Error* class and its subclasses represent abnormal conditions that should generally not be caught in your application. These indicate serious system problems, often JVM related, which an application might not be able to handle.

### Hierarchy of Error Class

The figure below illustrates the class hierarchy of Error Class:




**Learn more about below Java built-in errors:**

- **OutOfMemoryError**
- **StackOverflowError**
- **NoClassDefFoundError**

**2. Exception**

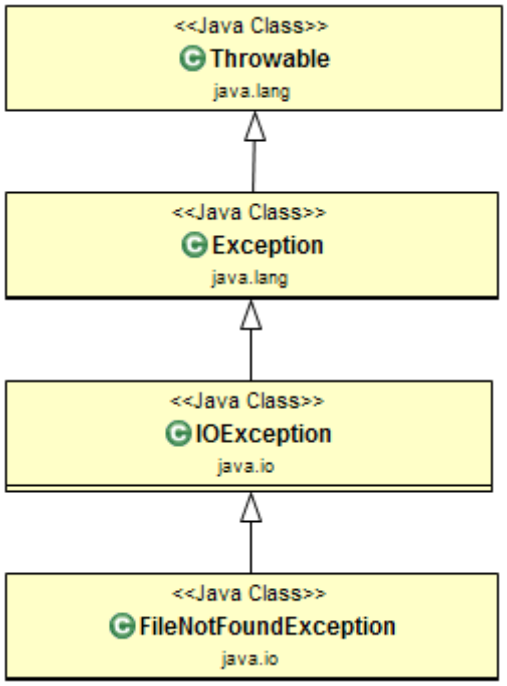
**Checked Exceptions**

These are exceptions that a method can throw but must either catch them or declare them using the *throws* keyword. They are direct subclasses of the *Exception* class, except *RuntimeException*.

**Java built-in checked exceptions:**

- **FileNotFoundException**
- **EOFException**
- **ClassNotFoundException**
- **SQLException**
- **NoSuchMethodException**
- **InterruptedException**

The figure below illustrates the class hierarchy of the Exception Class:




**Exception Class Example:** In this example, *FileReader* class try to read a file from an invalid location will throw *FileNotFoundException* exception.

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


        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader(new File("/invalid/file/location")));
        } catch (FileNotFoundException e) {
            System.err.println("FileNotFoundException caught!");
        }
    }
}
```

**Output:**

```
FileNotFoundException caught!
```

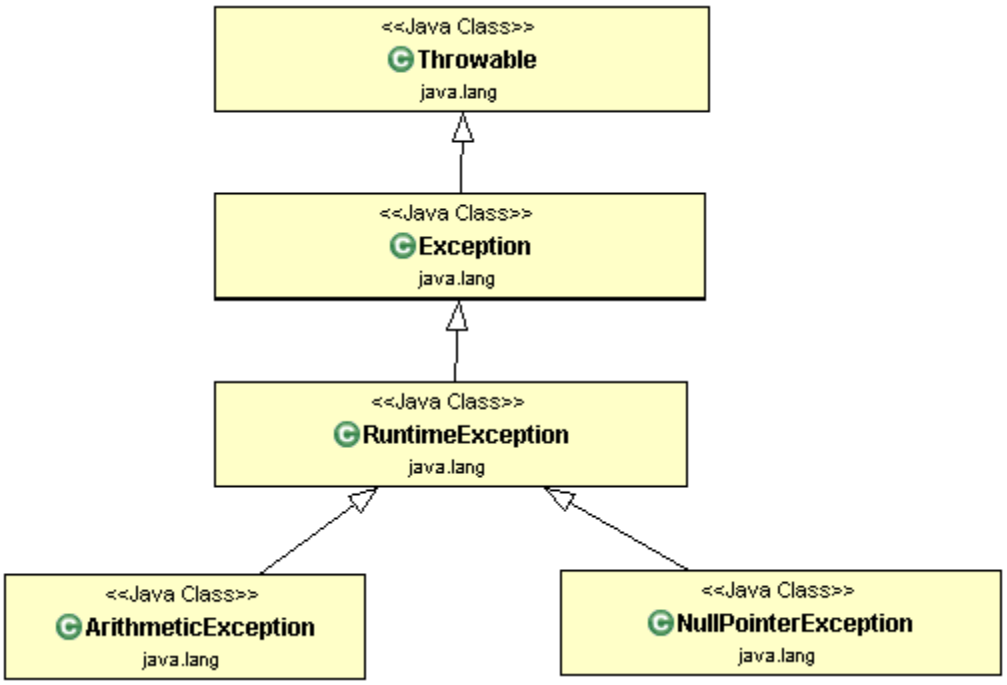
**RuntimeException (Unchecked Exceptions)**

These exceptions are not checked at compile-time. They're a direct subclass of the *RuntimeException* class.

**Java built-in unchecked exceptions:**

- **NullPointerException**
- **ArrayIndexOutOfBoundsException**
- **StringIndexOutOfBoundsException**
- **ArithmeticException**
- **IllegalArgumentException**
- **NumberFormatException**
- **IllegalStateException**
- **ClassCastException**

The figure below illustrates the class hierarchy of the *RuntimeException* Class:



**RuntimeException Example:** In the below example, the Person class object is not created using the new keyword but it is just declared with a null value. Now we are trying to access the personName field value on the null reference so JVM will throw a *NullPointerException* exception here.

```
public class NullPointerExceptionExample {

    public static void main(String[] args) {

        Person personObj = null;

        try {
            String name = personObj.personName; // Accessing the field of a null object
            personObj.personName = "Jon Doe"; // Modifying the field of a null object
        } catch (NullPointerException e) {
            System.err.println("NullPointerException caught!");
        }

    }
}

class Person {

    public String personName;

    public String getPersonName() {
        return personName;
```

```
    }

    public void setPersonName(String personName) {
        this.personName = personName;
    }

}
```

Output:

NullPointerException caught!

**Chained Exceptions**

Chained exceptions in Java allow you to relate one exception with another, forming a chain of exceptions. This is useful when an exception occurs as a direct result of another exception. Chained exceptions provide a way to capture the original cause of an exception and pass it along with the new exception. This mechanism helps debug and understand the root cause of an error.

**1. Overview of Chained Exceptions**

Chained exceptions allow you to keep track of the original cause of an exception. This is achieved by associating one exception with another using the Throwable class methods.

**2. Creating Chained Exceptions**

To create a chained exception, you need to pass the original exception as a parameter to the constructor of the new exception. This way, the new exception contains a reference to the original exception.

**Example:**

```
public class ChainedExceptionDemo {
    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void method1() throws Exception {
        try {
            method2();
        } catch (Exception e) {
            throw new Exception("Exception in method1", e);
        }
    }

    public static void method2() throws Exception {
        throw new Exception("Exception in method2");
    }
}
```

**Output:**

```
java.lang.Exception: Exception in method1
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:12)
    at ChainedExceptionDemo.main(ChainedExceptionDemo.java:5)
Caused by: java.lang.Exception: Exception in method2
```

```
    at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:17)
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:10)
    ... 1 more
```

### 3. Methods for Chaining Exceptions

Java provides several methods in the Throwable class to work with chained exceptions:

- Throwable(Throwable cause): Initializes a new throwable with the specified cause.
- Throwable(String message, Throwable cause): Initializes a new throwable with the specified detail message and cause.
- Throwable getCause(): Returns the cause of this throwable or null if the cause is nonexistent or unknown.
- Throwable initCause(Throwable cause): Initializes the cause of this throwable.

**Example:**

```
public class ChainedExceptionMethodsDemo {
    public static void main(String[] args) {
        try {
            methodA();
        } catch (Exception e) {
            System.out.println("Caught: " + e);
            System.out.println("Original cause: " + e.getCause());
        }
    }

    public static void methodA() throws Exception {
        try {
            methodB();
        } catch (Exception e) {
```

```
            Exception newException = new Exception("Exception in methodA");
            newException.initCause(e);
            throw newException;
        }
    }

    public static void methodB() throws Exception {
        throw new Exception("Exception in methodB");
    }
}
```

**Output:**

```
Caught: java.lang.Exception: Exception in methodA
Original cause: java.lang.Exception: Exception in methodB
```

**4. Example of Chained Exceptions**

Here's a more comprehensive example that demonstrates chained exceptions in a real-world scenario:

**Example Code:**

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class ChainedExceptionExample {
    public static void main(String[] args) {
        try {
            readFile("nonexistentfile.txt");
        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }

    public static void readFile(String fileName) throws Exception {
        try {
            FileReader fr = new FileReader(new File(fileName));
            fr.read();
        } catch (FileNotFoundException e) {
            throw new Exception("File not found", e);
        } catch (IOException e) {
            throw new Exception("I/O error while reading file", e);
        }
    }
}
```

**Output:**

```
java.lang.Exception: File not found
    at ChainedExceptionExample.readFile(ChainedExceptionExample.java:17)
    at ChainedExceptionExample.main(ChainedExceptionExample.java:10)
Caused by: java.io.FileNotFoundException: nonexistentfile.txt (No such file or directory)
    at java.base/java.io.FileInputStream.open0(Native Method)
    at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
    at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
    at java.base/java.io.FileReader.<init>(FileReader.java:75)
    at ChainedExceptionExample.readFile(ChainedExceptionExample.java:14)
```

```
    ... 1 more
```

**5. Handling Chained Exceptions**

When handling chained exceptions, you can use the getCause method to retrieve the original cause and take appropriate actions.

**Example:**

```
public class HandleChainedExceptions {
    public static void main(String[] args) {
        try {
            readFile("nonexistentfile.txt");
        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            Throwable cause = e.getCause();
            if (cause != null) {
                System.out.println("Original cause: " + cause.getMessage());
            }
        }
    }

    public static void readFile(String fileName) throws Exception {
        try {
            FileReader fr = new FileReader(new File(fileName));
            fr.read();
        } catch (FileNotFoundException e) {
            throw new Exception("File not found", e);
        } catch (IOException e) {
            throw new Exception("I/O error while reading file", e);
        }
    }
```

}

**Output:**

Caught exception: File not found

Original cause: nonexistentfile.txt (No such file or directory)

**6. Benefits of Chained Exceptions**

- **Preserves the Original Cause**: Chained exceptions preserve the original exception, making it easier to trace the root cause of an error.
- **Improves Debugging**: By maintaining the chain of exceptions, developers can understand the sequence of events that led to an error.
- **Enhances Error Reporting**: Chained exceptions provide more context, improving error reporting and logging.

## Try with resources

The *try-with-resources* statement in Java is a powerful feature introduced in Java 7. It is used to automatically manage resources such as files, database connections, sockets, etc. that need to be closed after they are no longer needed. This feature helps in writing cleaner and more reliable code by ensuring that resources are closed properly, avoiding resource leaks.

**1. What is try-with-resources?**

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.

**2. Benefits of try-with-resources**

- **Automatic Resource Management**: Automatically closes resources when they are no longer needed.
- **Simplified Code**: Reduces boilerplate code required to explicitly close resources.
- **Reduced Resource Leaks**: Ensures that resources are properly closed, reducing the risk of resource leaks.
- **Improved Readability**: Enhances the readability and maintainability of the code.

**3. Using try-with-resources**

To use the try-with-resources statement, you need to declare the resources within the try statement. The resources must implement the AutoCloseable interface, which includes the close() method.

**Syntax:**

```
try (ResourceType resource = new ResourceType()) {
    // Use the resource
} catch (ExceptionType e) {
    // Handle exceptions
}
```

**4. Example with FileReader and BufferedReader**

Let's consider an example where we read a file using FileReader and BufferedReader with the try-with-resources statement.

**Example:**

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
```

```
public class TryWithResourcesExample {
    public static void main(String[] args) {
        String filePath = "example.txt";

        try (FileReader fileReader = new FileReader(filePath);
             BufferedReader bufferedReader = new BufferedReader(fileReader)) {

            String line;
            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }

        } catch (IOException e) {
            System.err.println("Error reading the file: " + e.getMessage());
        }
    }
}
```

**Explanation:**

- FileReader and BufferedReader are declared within the try statement.
- Both resources are automatically closed at the end of the try block.
- The catch block handles any IOException that may occur during file reading.

**5. Example with Custom Resources**

You can also use try-with-resources with custom resources by implementing the AutoCloseable interface.

**Example:**

```
class CustomResource implements AutoCloseable {
```

```
    public void useResource() {
        System.out.println("Using custom resource");
    }

    @Override
    public void close() {
        System.out.println("Closing custom resource");
    }
}

public class CustomResourceExample {
    public static void main(String[] args) {
        try (CustomResource customResource = new CustomResource()) {
            customResource.useResource();
        }
    }
}
```

**Output:**

Using custom resource

Closing custom resource

**Explanation:**

- CustomResource implements the AutoCloseable interface.
- The close() method is overridden to define the resource cleanup logic.
- The custom resource is automatically closed at the end of the try block.

### 6. Handling Multiple Resources

You can declare multiple resources within the try statement, and they will be closed in the reverse order of their creation.

**Example:**

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;


public class MultipleResourcesExample {

    public static void main(String[] args) {

        String inputFilePath = "input.txt";

        String outputFilePath = "output.txt";


        try (FileReader fileReader = new FileReader(inputFilePath);

             BufferedReader bufferedReader = new BufferedReader(fileReader);

             FileWriter fileWriter = new FileWriter(outputFilePath)) {


            String line;

            while ((line = bufferedReader.readLine()) != null) {

                fileWriter.write(line + "\n");

            }


        } catch (IOException e) {

            System.err.println("Error handling the files: " + e.getMessage());

        }

    }
```

}

**Explanation:**

- FileReader, BufferedReader, and FileWriter are declared within the try statement.
- All resources are automatically closed in the reverse order of their creation.

**Steps to Create a Custom Exception in Java**

**1. Decide the Exception Type:**

Decide whether your exception should be checked or unchecked.

**Checked Exceptions:** These are the exceptions that a method can throw but cannot handle on its own. Any method that might throw a checked exception must either handle it using a **try-catch block** or declare it using the **throws** keyword.

**Unchecked Exceptions:** These are the exceptions that can be thrown during the execution of the program but were not anticipated. They extend *RuntimeException* and don't need to be declared or caught.

**2. Choose a Meaningful Name:**

Your custom exception name should typically end with "Exception" to maintain clarity and adhere to Java's naming convention.

**For example:**

ResourceNotFound**Exception**

BadRequest**Exception**

AgeValidator**Exception**

BlogAPI**Exception**

**3. Extend the Appropriate Exception Class:**

For checked exceptions, extend the *Exception* class.

For unchecked exceptions, extend the *RuntimeException* class.

**4. Create Constructors:**

To give more context or customization ability to your exception, create constructors for it. At a minimum, you should provide:

- A default constructor.
- A constructor that takes a string parameter for the error message.
- Additional constructors as required (e.g., to initialize custom fields).

**5. (Optional) Add Custom Fields and Methods:**

If you want your exception to hold additional information or provide specific functionalities, you can add custom fields and methods.

Now, let's see how to use the above step to create custom-checked exceptions and custom-unchecked exceptions.

**Create a Checked Custom Exception**

Let's create a checked custom exception named InvalidOrderException:

```
public class InvalidOrderException extends Exception {

    public InvalidOrderException() {
        super("The order is invalid");
    }

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

**Create a Custom Unchecked Exception**

```
public class OrderConfigurationException extends RuntimeException {

    public OrderConfigurationException() {
```

```
        super("Configuration of the order is incorrect");
    }

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

**Using Custom Exceptions**

You can throw your custom exception just like any other exception using the throw statement.

```
public class OrderProcessor {

    public void validateOrder(Order order) throws InvalidOrderException {
        if(order == null || order.getItems().isEmpty()) {
            throw new InvalidOrderException("Order is empty or null");
        }
    }

    public void configureOrder(Order order) {
        if(order.getConfiguration() == null) {
            throw new OrderConfigurationException("Order lacks necessary configuration");
        }
    }
}
```

**Usage**

For the checked exception (InvalidOrderException), you'll either need to catch it or declare that your method may throw it:

```
public static void main(String[] args) {
    OrderProcessor processor = new OrderProcessor();
    try {
        processor.validateOrder(null); // This will throw InvalidOrderException
    } catch (InvalidOrderException e) {
        System.out.println("Error: " + e.getMessage());
    }

    processor.configureOrder(new Order()); // This will throw OrderConfigurationException
}
```

**Best Practices to Create Custom Exceptions in Java**

**Descriptive Naming:** Ensure the name of the exception class clearly indicates its purpose. Always end with "Exception", e.g., *InvalidInputException*.

**Extend the Right Class:**

- For checked exceptions, extend the *Exception* class.
- For unchecked exceptions, extend the RuntimeException class.

**Provide Useful Constructors:** Offer standard constructors such as one with no parameters, one with a String message, and one that accepts a *Throwable* cause.

```
public class CustomException extends Exception {
    public CustomException() { super(); }
    public CustomException(String message) { super(message); }
    public CustomException(String message, Throwable cause) { super(message, cause); }
    public CustomException(Throwable cause) { super(cause); }
}
```

**Document Properly:** Use *JavaDoc* to detail when the custom exception might be thrown and explain any additional fields or context it provides.

**Make Custom Exceptions Immutable:** Once created, exceptions should be immutable to ensure thread safety. Avoid setters in exception classes.

**Add Relevant Information:** Your custom exception can have additional fields that provide more information about the error. For instance, an *OrderNotFoundException* might contain an *orderId* field.

These are concise but integral guidelines to follow when creating custom exceptions in Java. They ensure that your custom exceptions are robust, maintainable, and consistent with Java's exception-handling standards.