Java Exception Handling

1. Java Exceptions
2. Checked vs Unchecked Exception
3. Try Catch Block
4. Final, Finally and Finalize
5. Throw and Throws
6. Customized Exception Handling
7. Exception Handling with Method Overriding

Java Exceptions

Exception handling in Java is an effective mechanism for managing runtime errors to ensure the application's regular flow is maintained. Some Common examples of exceptions include ClassNotFoundException, IOException, SQLException, RemoteException, etc. By handling these exceptions, Java enables developers to create robust and fault-tolerant applications.

Example: Showing an arithmetic exception or we can say a divide by zero exception.

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        int n = 10;
        int m = 0;

        int ans = n / m;

        System.out.println("Answer: " + ans);
    }
}
Output:
Exception in thread "main" java.lang.ArithmeticException Create
breakpoint: / by zero at Main.main(Main.java:11)
```

Note: When an exception occurs and is not handled, the program terminates abruptly and the code after it, will never execute.

Example: The below Java program modifies the previous example to handle an ArithmeticException using try-catch and finally blocks and keeps the program running.

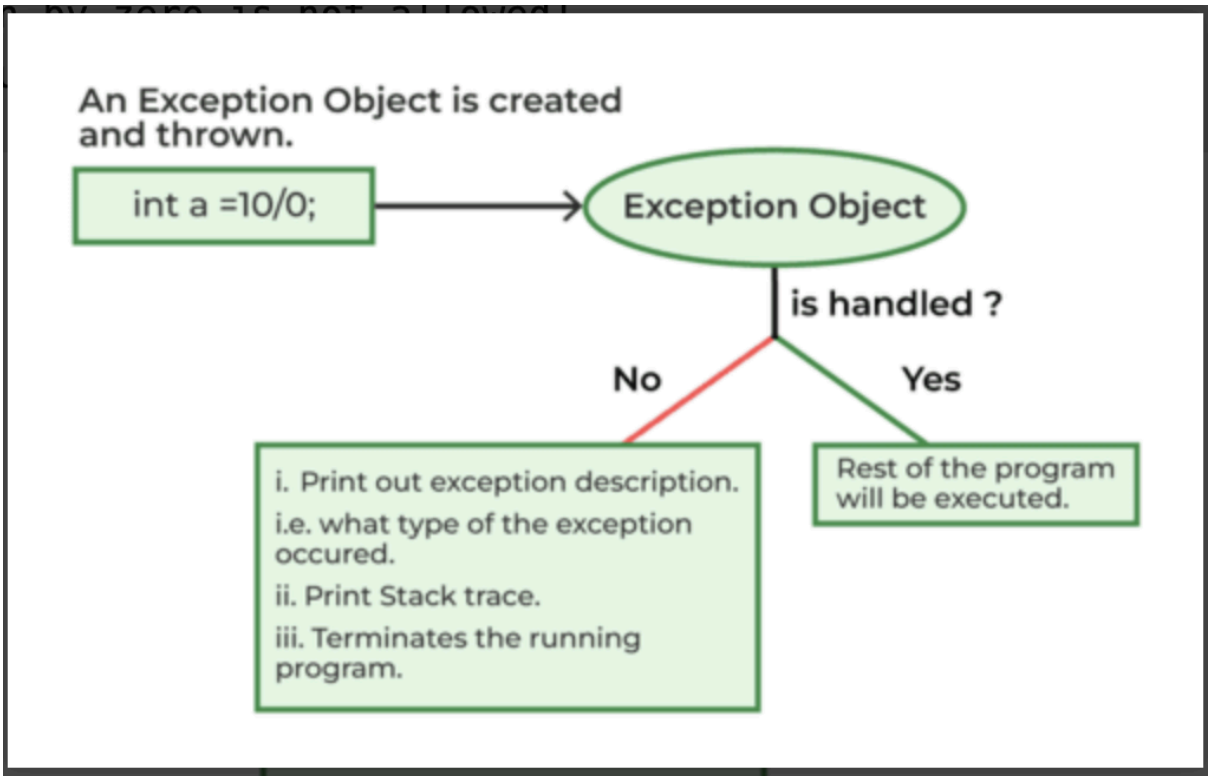
```
import java.io.*;

class Main {
    public static void main(String[] args){
        int n = 10;
        int m = 0;

        try {
            int ans = n / m;
            System.out.println("Answer: " + ans);
        } catch (ArithmeticException e) {
            System.out.println(
                "Error: Division by zero is not allowed!");
        }
```

```
        finally {
            System.out.println(
                "Program continues after handling the exception.");
        }
    }
}

Output
Error: Division by zero is not allowed!
Program continues after handling the exception.
```

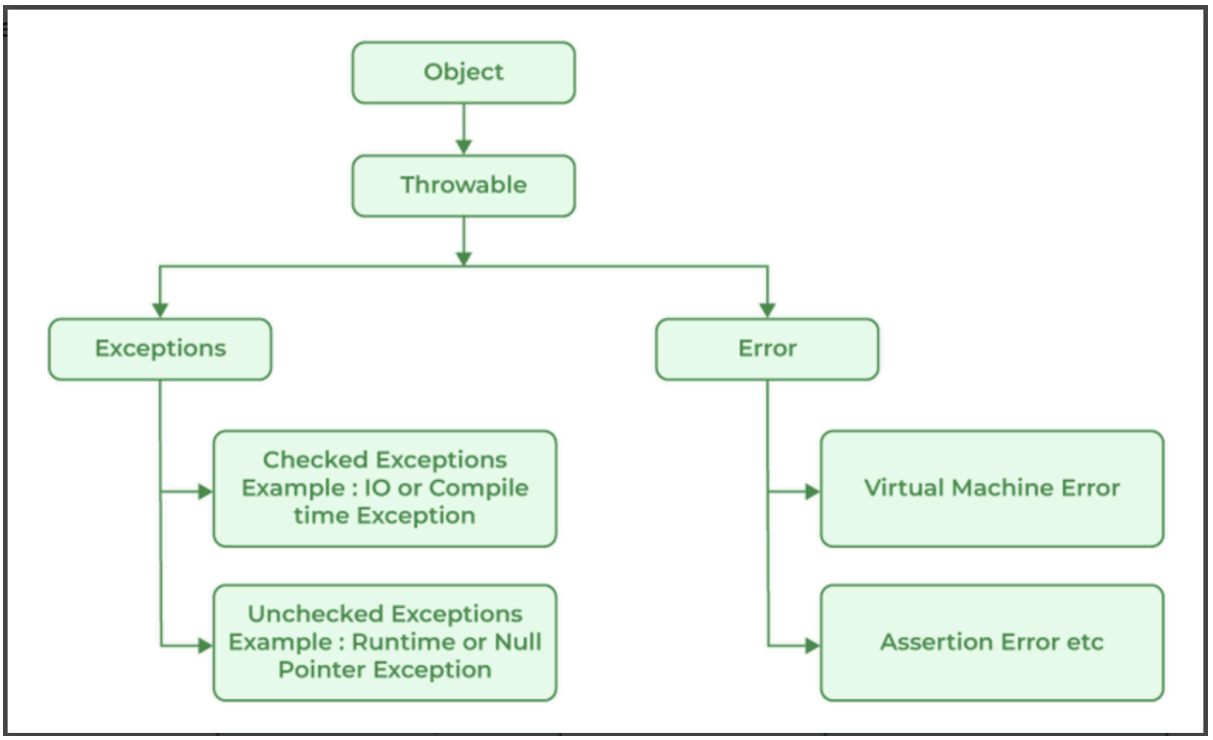


Java Exception Hierarchy

In Java, all exceptions and errors are subclasses of the Throwable class. It has two main branches

1. Exception.
2. Error

The below figure demonstrates the exception hierarchy in Java:



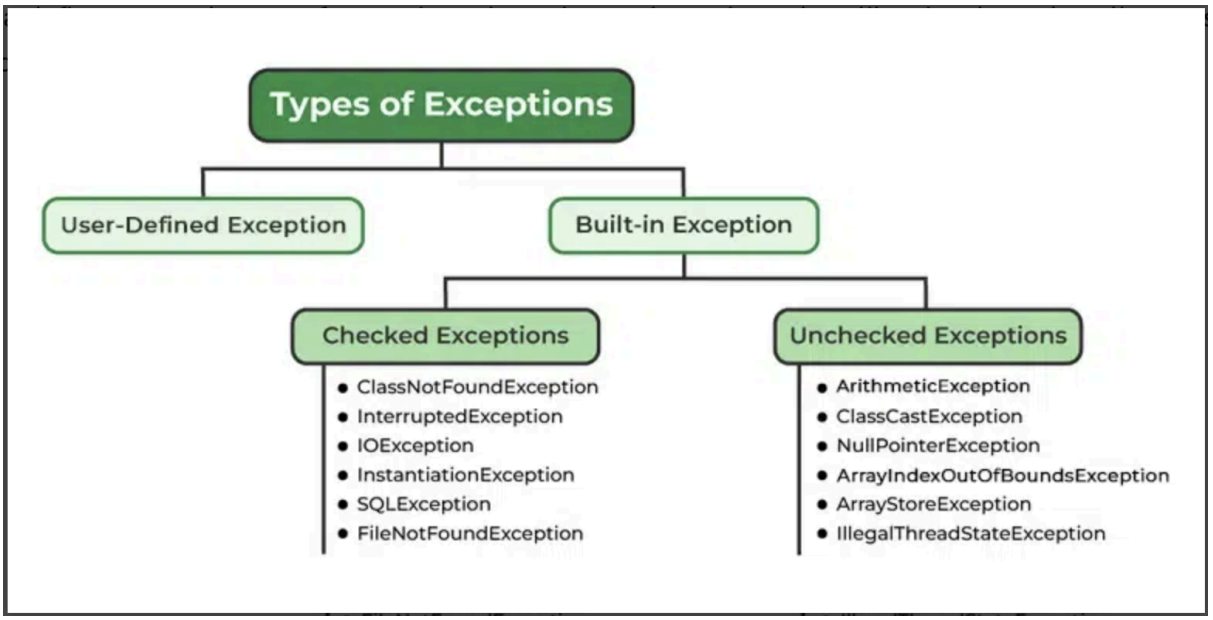
Major Reasons Why an Exception Occurs

Exceptions can occur due to several reasons, such as:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Out of bound
- Null reference
- Type mismatch
- Opening an unavailable file
- Database errors
- Arithmetic errors

Errors are usually beyond the control of the programmer and we should not try to handle errors.

Types of Java Exceptions
Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their exceptions.



Exceptions can be categorized in two ways:
1. Built-in Exceptions

- Checked Exception
- Unchecked Exception

2. user-defined Exceptions

1. Built-in Exception
Built-in Exceptions are pre-defined exception classes provided by Java to handle common errors during program execution. There are two types of built-in exceptions in java.

Checked Exceptions
Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler. Examples of Checked Exception are listed below:

- **ClassNotFoundException:** Throws when the program tries to load a class at runtime but the class is not found because it's belong not present in the correct location or it is missing from the project.
- **InterruptedException:** Thrown when a thread is paused and another thread interrupts it.
- **IOException**: Throws when input/output operation fails.
- InstantiationException: Thrown when the program tries to create an object of a class but fails because the class is abstract, an interface or has no default constructor.
- **SQLException:** Throws when there is an error with the database.

- **FileNotFoundException:** Thrown when the program tries to open a file that does not exist.

Unchecked Exceptions

The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception and even if we did not handle or declare it, the program would not give a compilation error. Examples of Unchecked Exception are listed below:

- **ArithmeticException:** It is thrown when there is an illegal math operation.
- **ClassCastException:** It is thrown when we try to cast an object to a class it does not belong to.
- **NullPointerException:** It is thrown when we try to use a null object (e.g. accessing its methods or fields).
- **ArrayIndexOutOfBoundsException:** This occurs when we try to access an array element with an invalid index.
- **ArrayStoreException:** This happens when we store an object of the wrong type in an array.
- **IllegalThreadStateException:** It is thrown when a thread operation is not allowed in its current state.

Checked vs Unchecked Exceptions

In Java, an exception is an unwanted or unexpected event that occurs during the execution of a program, i.e., at run time, that disrupts the normal flow of the program's instructions. In Java, there are two types of exceptions:

- **Checked Exception:** These exceptions are checked at compile time, forcing the programmer to handle them explicitly.
- **Unchecked Exception:** These exceptions are checked at runtime and do not require explicit handling at compile time.

Checked Exceptions in Java

Checked Exceptions are exceptions that are checked at compile time. If a method throws a checked Exception, then the exception must be handled using a try-catch block and declared the exception in the method signature using the throws keyword.

Types of Checked Exception

- **Fully Checked Exception:** A checked exception where all its child classes are also checked (e.g., IOException, InterruptedException).
- **Partially Checked Exception:** A checked exception where some of its child classes are unchecked (e.g., Exception).

Checked exceptions represent invalid conditions in areas outside the immediate control of the program like memory, network, file system, etc. Any checked exception is a subclass of Exception. Unlike unchecked exceptions, checked exceptions must be either caught by the caller or listed as part of the method signature using the throws keyword.

Example: Java Program to Illustrate Checked Exceptions Where FileNotFoundException occurs

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        // Getting the current root directory
        String root = System.getProperty("user.dir");
        System.out.println("Current root directory: " + root);

        // Adding the file name to the root directory
        String path = root + "\\message.txt";
        System.out.println("File path: " + path);

        // Reading the file from the path in the local directory
        FileReader f = new FileReader(path);
```

```
        // Creating an object as one of the ways of taking input
        BufferedReader b = new BufferedReader(f);

        for (int counter = 0; counter < 3; counter++)
            System.out.println(b.readLine());

        f.close();
    }
}
Output:
Main.java:17: error: unreported exception FileNotFo undException; must
be caught or declared to be throw
n
FileReader f= new FileReader(path);
A
Main.java:24: error: unreported exception IOExcepti on; must be caught
or declared to be thrown
System.out.println(b.readLine
Main.java:28: error: unreported exception IOExcepti on; must be caught
or declared to be thrown
f.close();
A
System.out.println(b.readLine());
Main.java:28: error: unreported exception IOExcepti on; must be caught
or declared to be thrown
f.close();
A
```

Checked Exceptions

To fix the above program, we either need to specify a list of exceptions using throws or we need to use a try-catch block. We have used throws in the below program. Since FileNotFoundException is a subclass of IOException, we can just specify IOException in the throws list and make the above program compiler-error-free.

Example: Handling Checked Exceptions

```
import java.io.*;

class Main {
    public static void main(String[] args) throws IOException {

        // Getting the current root directory
        String root = System.getProperty("user.dir");
        System.out.println("Current root directory: " + root);
```

```
        // Adding the file name to the root directory
        String path = root + "\\message.txt";
        System.out.println("File path: " + path);

        // Reading the file from the path in the local directory
        try {
            FileReader f = new FileReader(path);

            // Creating an object as one of the ways of taking input
            BufferedReader b = new BufferedReader(f);

            // Printing the first 3 lines of the file
            for (int counter = 0; counter < 3; counter++)
                System.out.println(b.readLine());

            f.close();
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("An I/O error occurred: " +
e.getMessage());
        }
    }
}
Output:
Current root directory: C:\pranoy\JavaCodes
File path: C:\pranoy\JavaCodes\message.txt
Hello
Test
How are you?
```

Explanation: In the above program we create a Java program which reads a file from the same directory. This program may throw exceptions like FileNotFoundException or IOException so we handle it using the try-catch block to handle the exceptions and execute the program without any interruption.

Unchecked Exceptions in Java

Unchecked exceptions are exceptions that are not checked at the compile time. In Java, exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.

Consider the following Java program. It compiles fine, but it throws an ArithmeticException when run. The compiler allows it to compile because ArithmeticException is an unchecked exception.

Example: Java program to illustrate the Runtime Unchecked Exception.

```
class Main {
    public static void main(String args[]) {
        // Here we are dividing by 0 which will not be caught at compile
time as there is no mistake but caught at runtime because it is
mathematically incorrect
        int x = 0;
        int y = 10;
        int z = y / x;
    }
}
Output:
Exception in thread "main" java.lang.ArithmeticException: / by zero at
Main.main(Main.java:11)
```

Note:

- Unchecked exceptions are runtime exceptions that are not required to be caught or declared in a throws clause.
- These exceptions are caused by programming errors, such as attempting to access an index out of bounds in an array or attempting to divide by zero.
- Unchecked exceptions include all subclasses of the RuntimeException class, as well as the Error class and its subclasses.

The separation into checked and unchecked exceptions sounded like a good idea at the time. Over the years, it has introduced more boilerplate and less aesthetically pleasing code patterns than it solved real problems. The typical pattern within the Java ecosystem is to hide the checked exception within an unchecked one.

Example:

```
try {
// Some I/O operation here
} catch( final IOException ex ) {
    throw new RuntimeException( "I/O operation failed", ex );
}
```

Difference Between Checked and Unchecked Exceptions

Checked Exception	Unchecked Exception
Checked exceptions are checked at compile time.	Unchecked exceptions are checked at run time.
Derived from Exception.	Derived from RuntimeException.
Caused by external factors like file I/O and database connection cause the checked Exception.	Caused by programming bugs like logical errors cause unchecked Exceptions.
Checked exceptions must be handled using a try-catch block or must be declared using the throws keyword	No handling is required.
Examples: IOException, SQLException, FileNotFoundException.	Examples: NullPointerException, ArrayIndexOutOfBoundsException.

Java Try Catch Block

A try-catch block in Java is a mechanism to handle exceptions. This make sure that the application continues to run even if an error occurs. The code inside the try block is executed, and if any exception occurs, it is then caught by the catch block.

Example: Here, we are going to handle the ArithmeticException using a simple try-catch block.

```
import java.io.*;

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

            // This will throw an ArithmeticException
            int res = 10 / 0;
        }
        // Here we are Handling the exception
        catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e);
        }

        // This line will executes weather an exception
        // occurs or not
        System.out.println("I will always execute");
    }
}

Output
Exception caught: java.lang.ArithmeticException: / by zero
I will always execute
```

Syntax of try Catch Block

```
try {

 // Code that might throw an exception

} catch (ExceptionType e) {

 // Code that handles the exception

}
```

1. try in Java

The try block contains a set of statements where an exception can occur.

```
try
{
    // statement(s) that might cause exception
}
```

2. catch in Java

The catch block is used to handle the uncertain condition of a try block. A try block is always followed by a catch block, which handles the exception that occurs in the associated try block.

```
catch
{
   // statement(s) that handle an exception
   // examples, closing a connection, closing
   // file, exiting the process after writing
   // details to a log file.
}
```

Internal working of try-catch Block

- Java Virtual Machine starts executing the code inside the try block.
- If an exception occurs, the remaining code in the try block is skipped, and the JVM starts looking for the matching catch block.
- If a matching catch block is found, the code in that block is executed.
- After the catch block, control moves to the finally block (if present).
- If no matching catch block is found the exception is passed to the JVM default exception handler.
- The final block is executed after the try catch block. regardless of whether an exception occurs or not.

```
try{
    int ans = 10/0;
} catch(ArithmeticException e) {
    System.out.println("caught ArithmeticException");
} finally {
    System.out.println("I will always execute whether an Exception occur
or not");
}
```

Example: Here, we demonstrate the working try catch block with multiple catch statements.

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        try {
            int res = 10 / 0;
            String s = null;
            System.out.println(s.length());
        } catch (ArithmeticException e) {
            System.out.println("Caught ArithmeticException: " + e);
        } catch (NullPointerException e) {
            System.out.println("Caught NullPointerException: " + e);
        }
    }
}

Output
Caught ArithmeticException: java.lang.ArithmeticException: / by zero
```

Example: Here, we demonstrate the working of nested try catch blocks.

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        try {
            System.out.println("Outer try block started");
            try {
                int n = 10;
                int res = n / 0;
            } catch (ArithmeticException e) {
                System.out.println("Caught ArithmeticException in inner
try-catch: " + e);
            } try {
                String s = null;
                System.out.println(s.length());
            } catch (NullPointerException e) {
                System.out.println("Caught NullPointerException in inner
try-catch: " + e);
            }
        } catch (Exception e) {
            System.out.println("Caught exception in outer try-catch: " +
e);
        } finally {
```

```
            System.out.println("Finally block executed");
        }
    }
}
Output:
Outer try block started
Caught ArithmeticException in inner try-catch:
java.lang.ArithmeticException: / by zero
Caught NullPointerException in inner try-catch:
java.lang.NullPointerException: Cannot invoke "String.length()" b
cause "<locall>" is null
Finally block executed
```

Java final, finally and finalize

In Java, the keywords "final", "finally" and "finalize" have distinct roles. final enforces immutability and prevents changes to variables, methods or classes. finally ensures a block of code runs after a try-catch, regardless of exceptions. finalize is a method used for cleanup before an object is garbage collected. Now, we will discuss each keyword in detail one by one.

final Keyword

The final keyword in Java is used with variables, methods and also with classes to restrict modification.

Syntax

```
// Constant value
final int b = 100;
```

Example: The below Java program demonstrates the value of the variable cannot be changed once initialized.

```
class A {
    public static void main(String[] args) {
        // Non final variable
        int a = 5;

        // final variable
        final int b = 6;

        // modifying the non final variable
        a++;

        // modifying the final variable Immediately gives Compile Time
error
        b++;
    }
}
Output:
Java: cannot assign a value to final variable b
```

Note: If we declare any variable as final, we can't modify its value. Attempting to do so results in a compile-time error.

In the above example, we used the final keyword to declare a variable that cannot be modified after its initialization. Similarly, the final keyword can also be applied to methods and classes in Java to impose certain restrictions.

finally Keyword

The finally keyword in Java is used to create a block of code that always executes after the try block, regardless of whether an exception occurs or not.

Syntax

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

Example: The below Java program demonstrates the working of finally block in exception handling.

```
public class Main {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block");
            int result = 10 / 0; // This will cause an exception
        }
        catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
        finally {
            System.out.println("finally block always execute");
        }
    }
}
Output:
Inside try block
Exception caught: / by zero
finally block always execute
```

Explanation: The try block attempts a division by zero, causing an ArithmeticException. The finally block executes, whether an exception occurs, ensuring cleanup or mandatory code execution.

finalize() Method

The finalize() method is called by the Garbage Collector just before an object is removed from memory. It allows us to perform clean up activities. Once the finalize() method completes, Garbage Collector destroys that object.finalize method is present in the Object class.

Syntax:

```
protected void finalize() throws Throwable{}
```

Note: finalize() is deprecated in Java 9 and should not be used in modern applications. It's better to use try-with-resources or other cleanup mechanisms instead of relying on finalize().

Example: The below Java program demonstrates the working of finalize() method in the context of garbage collection.

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Main g = new Main();

        System.out.println("Hashcode is: " + g.hashCode());

        // Making the object eligible for garbage collection
        g = null;

        System.gc();

        // Adding a short delay to allow GC to act
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("End of the garbage collection");
    }

    // Defining the finalize method
    @Override
    protected void finalize() {
        System.out.println("Called the finalize() method");
    }
}
Output:
Hashcode is: 1480010240
Called the finalize() method End of the garbage collection
```

Explanation: In the above example, an object "g" is created and its hash code is printed. The object is made eligible for garbage collection by setting it to null and invoking System.gc().

final vs finally vs finalize

The table below illustrates the differences between "final", "finally" and "finalize."

final	finally	finalize
The **final** keyword applies restrictions on variables, methods and classes.	The **finally** block in exception handling is used with try-catch blocks.	finalize is a method of object class
Prevent modification of variables, inheritance of classes or overriding of methods.	The code that is written inside the finally block is always executed after the try-catch block whether an exception occurs or not .	finalize method in Java is used to perform cleanup operations before an object is garbage collected.
Variables, methods and classes.	Only within a try-catch block.	Objects, specifically by overriding the method in a class
Executes when declared	Always executed after try-catch block.	Called by the garbage collector when an object is about to be deleted, but it's not guaranteed to run.

throw and throws in Java

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.

Java 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.

Syntax:

```
throw Instance
```

Where instance is an object of type Throwable (or its subclasses, such as Exception).

Example:

```
throw new ArithmeticException("/ by zero");
```

But this exception i.e., Instance must be of type Throwable or a subclass of Throwable.

Example: This example demonstrates where an exception is thrown, caught and rethrown inside a method.

```
class Main {
    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.
```

Explanation: The above example demonstrates the use of the throw keyword to explicitly throw a NullPointerException. The exception is caught inside the fun() method and rethrown, where it is then caught in the main() method.

Example: This example demonstrates an arithmetic exception.

```
class Main {
    public static void main(String[] args){
        int numerator = 1;
        int denominator = 0;

        if (denominator == 0) {
            // Manually throw an ArithmeticException
            throw new ArithmeticException("Cannot divide by zero");
        } else {
            System.out.println(numerator / denominator);
        }
    }
}
Output:
Exception in thread "main" java.lang.ArithmeticException: Cannot divide
by zero at Main.main(Main.java:9)
```

Explanation: The above example demonstrates an exception using throw, where an ArithmeticException is explicitly thrown due to division by zero.

Java 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.

In a program, if there is a chance of raising an exception then the compiler always warns us about it and we must handle that checked exception. Otherwise, we will get a compile time error saying unreported exception XXX must be caught or declared to be thrown. To prevent this compile time error we can handle the exception in two ways:

- By using try catch
- By using the throws keyword

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.

Example 1: Unhandled Exception

```
class Main {
    public static void main(String[] args) {
        Thread.sleep(10000);
        System.out.println("Hello Learners");
    }
}
Output:
error: unreported exception InterruptedException; must be caught or
declared to be thrown
```

Explanation: In the above program, we are getting compile time errors because there is a chance of exception if the main thread is going to sleep, other threads get the chance to execute the main() method which will cause an InterruptedException.

Example 2: Using throws to Handle Exception

```
class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread.sleep(10000);
        System.out.println("Hello Learners");
    }
}
Output:
Hello Learners
```

Explanation: In the above program, by using the throws keyword we handled the InterruptedException and we will get the output as Hello Learners.

Example 3: Throwing an Exception with throws

```
class Main {
    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.
```

Explanation: The above example throws a IllegalAccessException from a method and handles it in the main method using a try-catch block.

Difference Between throw and throws

The main differences between throw and throws in Java are as follows:

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 is used in the method signature.
It can throw both checked and unchecked exceptions.	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 {}

User-Defined Custom Exception in Java

Java provides us the facility to create our own exceptions by extending the Java Exception class. Creating our own Exception is known as a custom exception in Java or a user-defined exception in Java. In simple words, we can say that a User-Defined Custom Exception or custom exception is creating your own exception class and throwing that exception using the "throw" keyword. Now, before understanding the concept in depth, let's go through the example below:

Example: In this example, a custom exception MyException is created and thrown in the program.

```
// A Class that represents user-defined exception
class MyException extends Exception {
    public MyException(String m) {
        super(m);
    }
}

// A Class that uses the above MyException
public class SetText {
    public static void main(String args[]) {
        try {
            // Throw an object of user-defined exception
            throw new MyException("This is a custom exception");
        } catch (MyException ex) {
            System.out.println("Caught");
            System.out.println(ex.getMessage());
        }
    }
}

Output
Caught
This is a custom exception
```

Java Custom Exception

A custom exception in Java is an exception defined by the user to handle specific application requirements. These exceptions extend either the Exception class (for checked exceptions) or the RuntimeException class (for unchecked exceptions).

Why Use Java Custom Exceptions?

- To represent application-specific errors.
- To add clear, descriptive error messages for better debugging.
- To encapsulate business logic errors in a meaningful way.
- Types of Custom Exceptions

Types of custom exceptions in Java:

1. **Checked Exceptions:** It extends the Exception class. and it must be declared in the throws clause of the method signature.
2. **Unchecked Exceptions:** It extends the RuntimeException class.

Create a User-Defined Custom Exception

1. Create a new class that extends Exception (for checked exceptions) or RuntimeException (for unchecked exceptions).
2. Provide constructors to initialize the exception with custom messages.
3. Add methods to provide additional details about the exception. (this is optional)

Example 1: Checked Custom Exception (Real-World Scenario)

```
// Custom Checked Exception
class InvalidAgeException extends Exception {
    public InvalidAgeException(String m) {
        super(m);
    }
}

// Using the Custom Exception
public class Main {
    public static void validate(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or above.");
        }
        System.out.println("Valid age: " + age);
    }

    public static void main(String[] args) {
        try {
            validate(12);
        } catch (InvalidAgeException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}
Output
Caught Exception: Age must be 18 or above.
```

Explanation: The above example defines a custom checked exception InvalidAgeException that is thrown when an age is below 18. The validate() method checks the age and throws the exception if the age is invalid. In the main() method, the exception is caught and the error message is printed.

Example 2: Unchecked Custom Exception

```
// Custom Unchecked Exception
class DivideByZeroException extends RuntimeException {
    public DivideByZeroException(String m) {
        super(m);
    }
}

// Using the Custom Exception
public class Main {
    public static void divide(int a, int b) {
        if (b == 0) {
            throw new DivideByZeroException("Division by zero is not
allowed.");
        }
        System.out.println("Result: " + (a / b));
    }

    public static void main(String[] args) {
        try {
            divide(10, 0);
        } catch (DivideByZeroException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}

Output
Caught Exception: Division by zero is not allowed.
```

Explanation: The above example defines a custom unchecked exception DivideByZeroException that is thrown when we are trying to divide by zero. The divide() method checks if the denominator is zero and throws the exception if true. In the main() method, the exception is caught and the error message is printed.

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. It ensures that the overridden method in the subclass does not declare broader or new checked exceptions than those declared by the superclass method, maintaining consistency and type safety during runtime execution

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.

Example: Method Overriding with Unchecked Exception

```
// Superclass without exception declaration
class SuperClass {
    void method() {
        System.out.println("SuperClass method executed");
    }
}

// Subclass declaring an unchecked exception
class SubClass extends SuperClass {
    @Override
    void method() throws ArithmeticException {
        System.out.println("SubClass method executed");
        throw new ArithmeticException("Exception in SubClass");
    }

    public static void main(String[] args) {
        SuperClass s = new SubClass();
        try {
            s.method();
        } catch (ArithmeticException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}

Output
SubClass method executed
Caught Exception: Exception in SubClass
```

Explanation: In this example, the SuperClass method does not declare any exceptions. The SubClass method overrides the method() and declares an unchecked exception i.e. the ArithmeticException. The main method demonstrates how the exception is thrown and caught in the SubClass.

Method Overriding with Checked Exception Rules
When exception handling is involved with method overriding, ambiguity occurs. The compiler gets confused as to which definition is to be followed. There are two types of problems associated with it which are as follows:

- **Problem 1:** If the superclass does not declare an exception.
- **Problem 2:** If the superclass declares an exception.

Let us discuss different cases under these problems and perceive their outputs.

Problem 1: Superclass Doesn't Declare an Exception
In this problem, two cases that will arise are as follows:

Case 1: If Superclass doesn't declare any exception and subclass declares checked exception.

```
import java.io.*;

class SuperClass {
  // SuperClass doesn't declare any exception
  void method() {
    System.out.println("SuperClass");
  }
}

// SuperClass inherited by the SubClass
class SubClass extends SuperClass {
  // method() declaring Checked Exception IOException
  void method() throws IOException {
    // IOException is of type Checked Exception so the compiler will
give Error
    System.out.println("SubClass");
  }

  public static void main(String args[]) {
    SuperClass s = new SubClass();
    s.method();
  }
}
```

```
Output:
GFG.java:20: error: method() in SubClass cannot override method() in
SuperClass
void method() throws IOException {
A
overridden method does not throw IOException
1 error
```

Explanation: A subclass cannot introduce a new checked exception if the superclass method does not declare any.

Case 2: If Superclass doesn't declare any exception and Subclass declares an unchecked exception.

```
class SuperClass {
    // SuperClass doesn't declare any exception
    void method() {
        System.out.println("SuperClass");
    }
}

// SuperClass inherited by the SubClass
class SubClass extends SuperClass {

    // method() declaring Unchecked  Exception ArithmeticException
    void method() throws ArithmeticException {
        // ArithmeticException is of type Unchecked Exception
        System.out.println("SubClass");
    }

    public static void main(String args[]) {
        SuperClass s = new SubClass();
        s.method();
    }
}

Output
SubClass
```

Explanation: ArithmeticException is unchecked, so the compiler allows it.

Problem 2: Superclass Declares an Exception
If the Superclass declares an exception. In this problem 3 cases will arise as follows:

Case 1: If Superclass declares an exception and Subclass declares exceptions other than the child exception of the Superclass declared exception.

```
class SuperClass {
    void method() throws RuntimeException {
        System.out.println("SuperClass");
    }
}

// Subclass declares an unrelated exception
class SubClass extends SuperClass {

    @Override
    void method() throws Exception {
        // Exception is not a child of RuntimeException
        System.out.println("SubClass");
    }

    public static void main(String[] args) {
        SuperClass o = new SubClass();
        o.method();
    }
}
Output:
GFG.java:18: error: method() in SubClass cannot override method() in
SuperClass
void method() throws Exception {
A
overridden method does not throw Exception
1 error
```

Case 2: If Superclass declares an exception and Subclass declares a child exception of the Superclass declared exception.

```
import java.io.*;

class SuperClass {
    // SuperClass declares an exception
    void method() throws RuntimeException {
        System.out.println("SuperClass");
    }
}
```

```
// SuperClass inherited by the SubClass
class SubClass extends SuperClass {
    // SubClass declaring a child exception of RuntimeException
    void method() throws ArithmeticException {
        System.out.println("SubClass");
    }

    public static void main(String args[])
    {
        SuperClass s = new SubClass();
        s.method();
    }
}

Output
SubClass
```

Case 3: If Superclass declares an exception and Subclass declares without exception.

```
import java.io.*;

class SuperClass {
    // SuperClass declares an exception
    void method() throws IOException {
        System.out.println("SuperClass");
    }
}

class SubClass extends SuperClass {
    // SubClass declaring without exception
    void method() {
        System.out.println("SubClass");
    }

    public static void main(String args[]) {
        SuperClass s = new SubClass();
        try {
             s.method();
        } catch (IOException e) {
             e.printStackTrace();
        }
    }
}
Output
SubClass
```

- **Case 1:** The subclass cannot declare an exception unrelated to the superclass exception.
- **Case 2:** The subclass can declare a child exception of the superclass exception.
- **Case 3:** The subclass can choose to declare no exception.