Exception Handling in Java

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained. Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc Advantage of Exception Handling The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application that is why we use exception handling.

Let's take a scenario:

statement 1;

statement 2;

statement 3;

statement 4;0

statement 5; //exception occurs

statement 6;

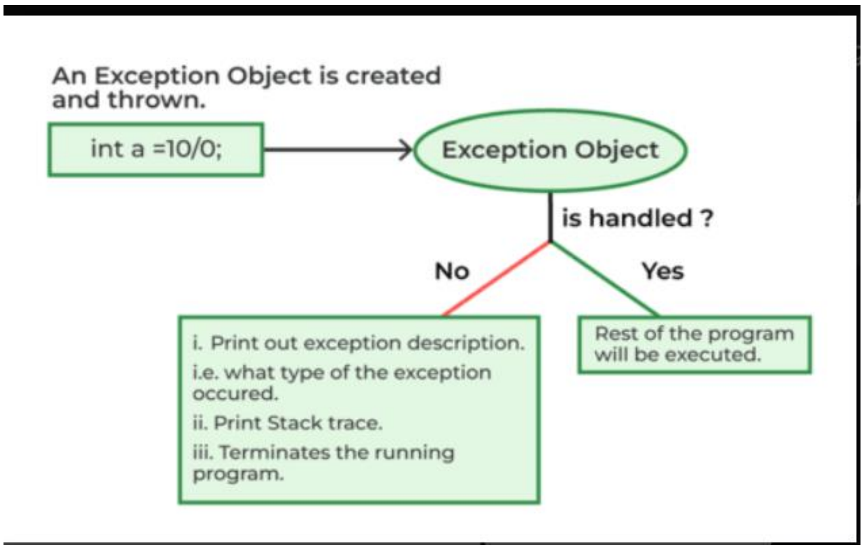
statement 7;

statement 8;

statement 9;

statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.


An Exception Object is created and thrown.
int a =10/0;
Exception Object
is handled ?
No
Yes
i. Print out exception description. i.e. what type of the exception occured.
ii. Print Stack trace.
iii. Terminates the running program.
Rest of the program will be executed.

Pictorial example for exception handling

Types of Java Exceptions

Exceptions can be categorized in two ways:

1. Built-inExceptions

   - Checked Exception
   - Unchecked Exception

2. User Defined Exception

1. Built-in Exception

Build-in Exception are pre-defined exception classes provided by Java to handle common errors during program execution.

1.1 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:

1. **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.
2. **InterruptedException:** Thrown when a thread is paused and another thread interrupts it.

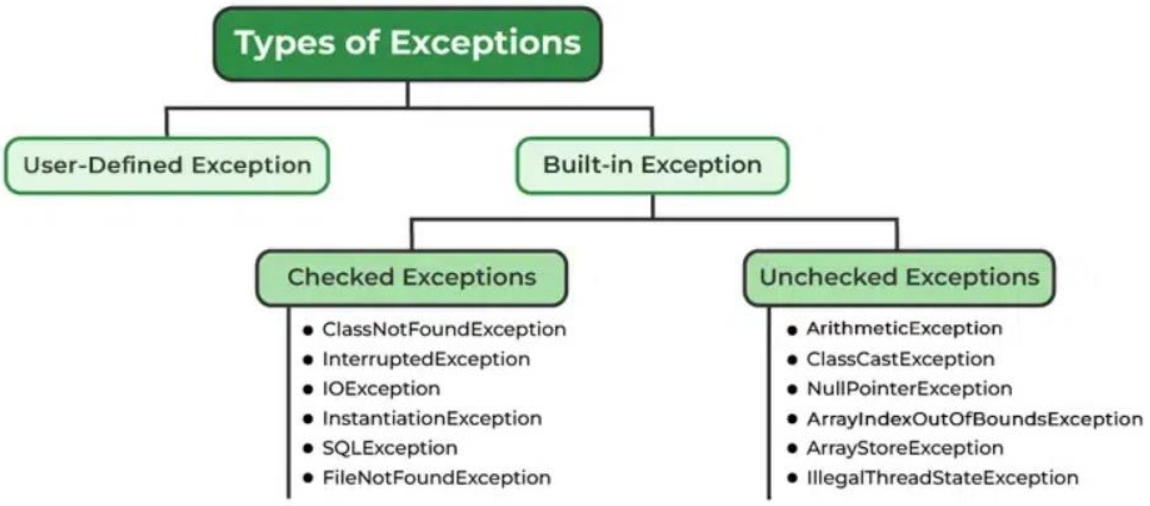
3. **IOException:** Throws when input/output operation fails
4. **InstantiationException:** Thrown when the program tries to create an object of a class but fails because the class is abstract, an interface, or has no default constructor.
5. **SQLException:** Throws when there's an error with the database.
6. **FileNotFoundException:** Thrown when the program tries to open a file that doesn't exist

1.2 Unchecked Exceptions

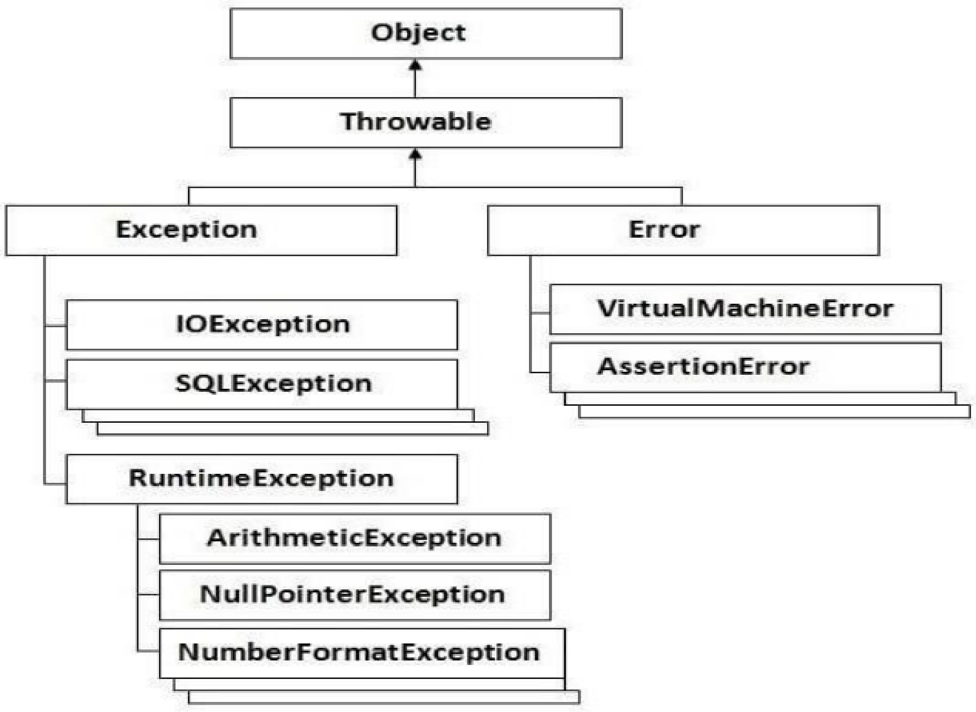
The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Examples of Unchecked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

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

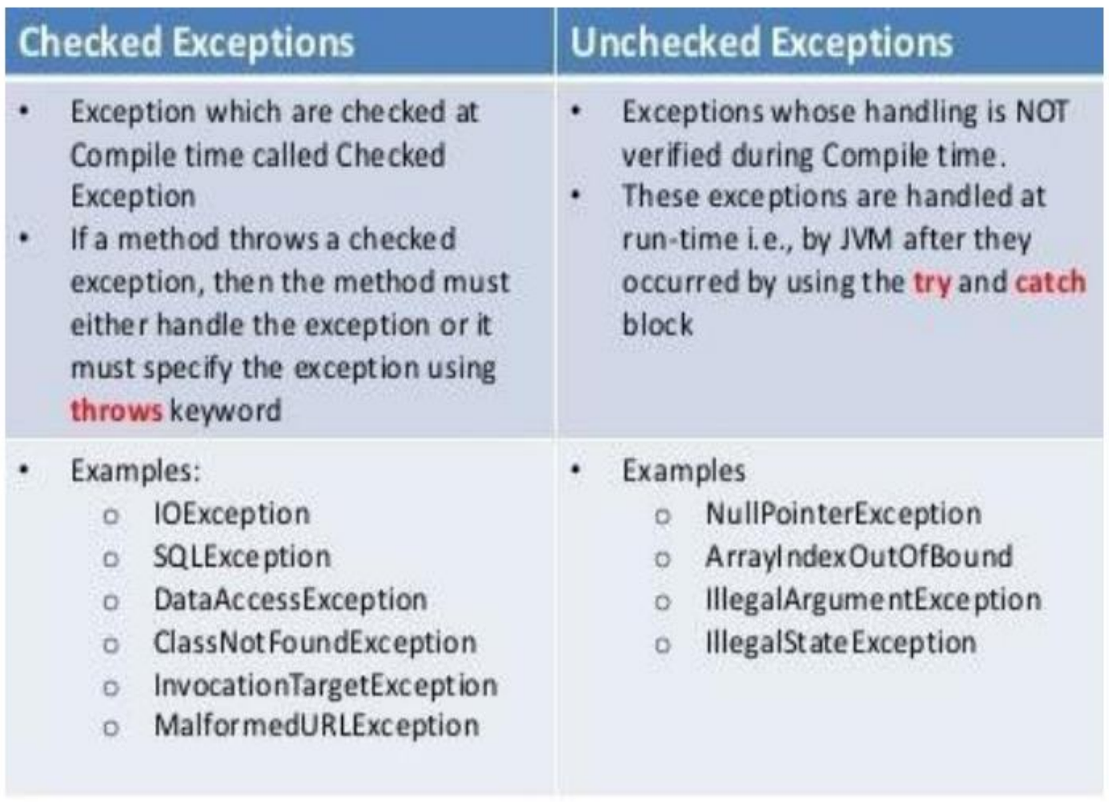
3. **User Defined Exceptions.**
   Sometimes, the built in exceptions in Java are not able to describe a certain situation. Such cases, users can also create exceptions which are called used defined Exceptions



Hierarchy of Java Exception classes



Checked and UnChecked Exceptions



Checked Exceptions	Unchecked Exceptions
• Exception which are checked at Compile time called Checked Exception • If a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword	• Exceptions whose handling is NOT verified during Compile time. • These exceptions are handled at run-time i.e., by JVM after they occurred by using the try and catch block
• Examples: o IOException o SQLException o DataAccessException o ClassNotFoundException o InvocationTargetException o MalformedURLException	• Examples o NullPointerException o ArrayIndexOutOfBound o IllegalArgumentException o IllegalStateException

Use of try,catch, finally, throw, throws in Exception Handling

try-catch block in Java is a mechanism to handle exceptions. This ensures 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.

```
import java.io.*;

class ABC {
    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

```
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 weather an exception occurs or not.

Example

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

Example:The working try catch block with multiple catch statements

```
// Java Program to Demonstrate try catch block
// with multiple catch statements
import java.util.*;

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

            // ArithmeticException
            int res = 10 / 0;

            // NullPointerException
            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 The working of ***nested try catch block.***

```
import java.util.*;

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

            // Outer try block
            System.out.println("Outer try block started");

            try {
                // Inner try block 1
                int n = 10;
                int res = n / 0;  // This will throw ArithmeticException
            } catch (ArithmeticException e) {
                System.out.println
                  ("Caught ArithmeticException in inner try-catch: " + e);
            }

            try {

                // Inner try block 2
                String s = null;
                System.out.println(s.length());  // This will throw NullPointerException
            } catch (NullPointerException e) {
                System.out.println
                  ("Caught NullPointerException in inner try-catch: " + e);
            }
        } catch (Exception e) {

            // Outer catch block
            System.out.println
              ("Caught exception in outer try-catch: " + e);
        } finally {

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

OUTPUT

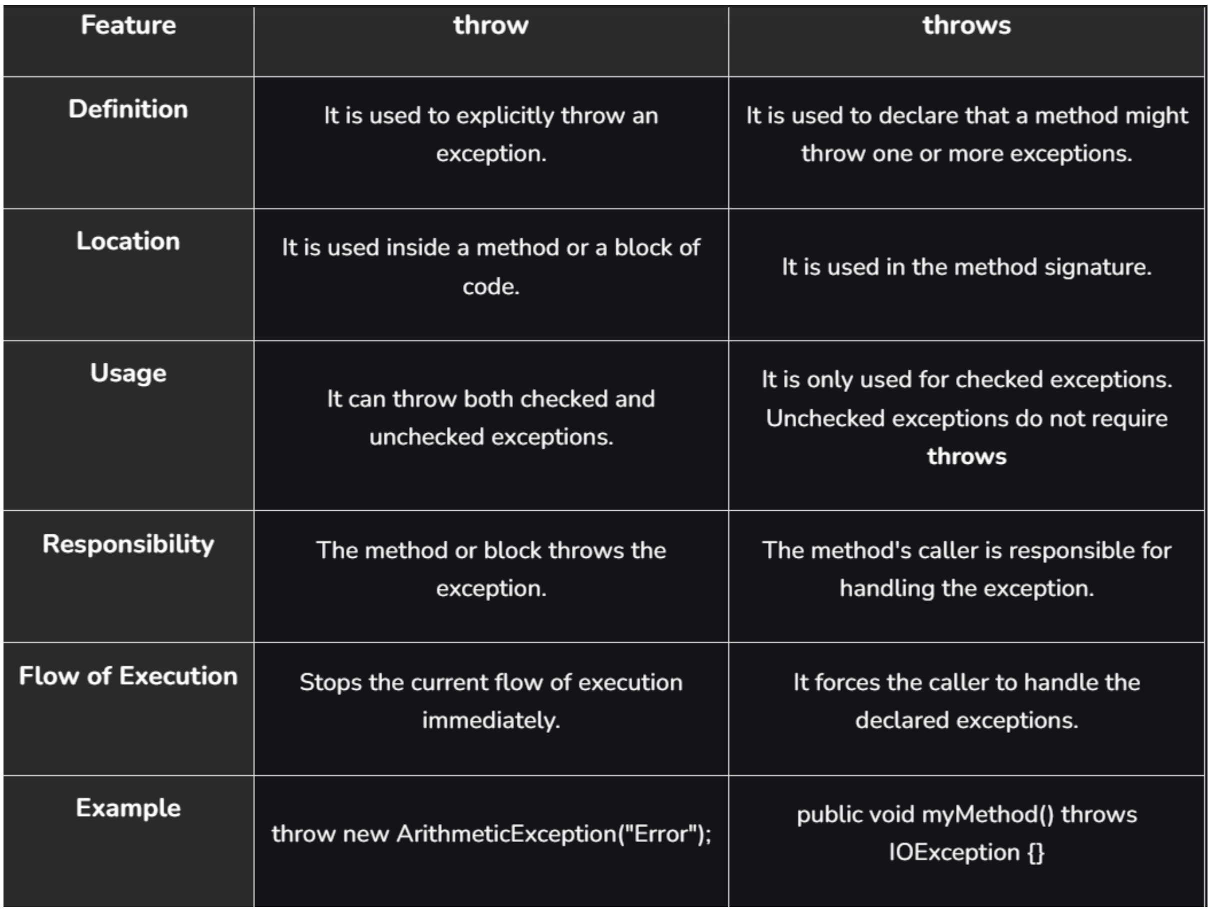
```
Outer try block started
Caught ArithmeticException in inner try-catch: java.lang.ArithmeticException: /
by zero
Caught NullPointerException in inner try-catch: java.lang.NullPointerException
Finally b...
```

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 run time error such
as **NullPointerException**, **ArrayIndexOutOfBoundsException**, etc. To handle these errors effectively, Java provides two key
concepts, **throw** and **throws**.

Difference Between throw and throws

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



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

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 exception. The throw keyword is mainly used to throw custom exceptions.

Syntax of throw in Java

throw Instance

Where instance is an object of type Throwable

Example:

throw new ArithmeticException("/ by zero");

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

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.

```
// Java program to demonstrate
// how to throw an exception
class abc {
    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
```

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 of Java throws

```
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 compulsorily we should handle that checked exception, Otherwise, we will get 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:

1. By using try catch
2. 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.

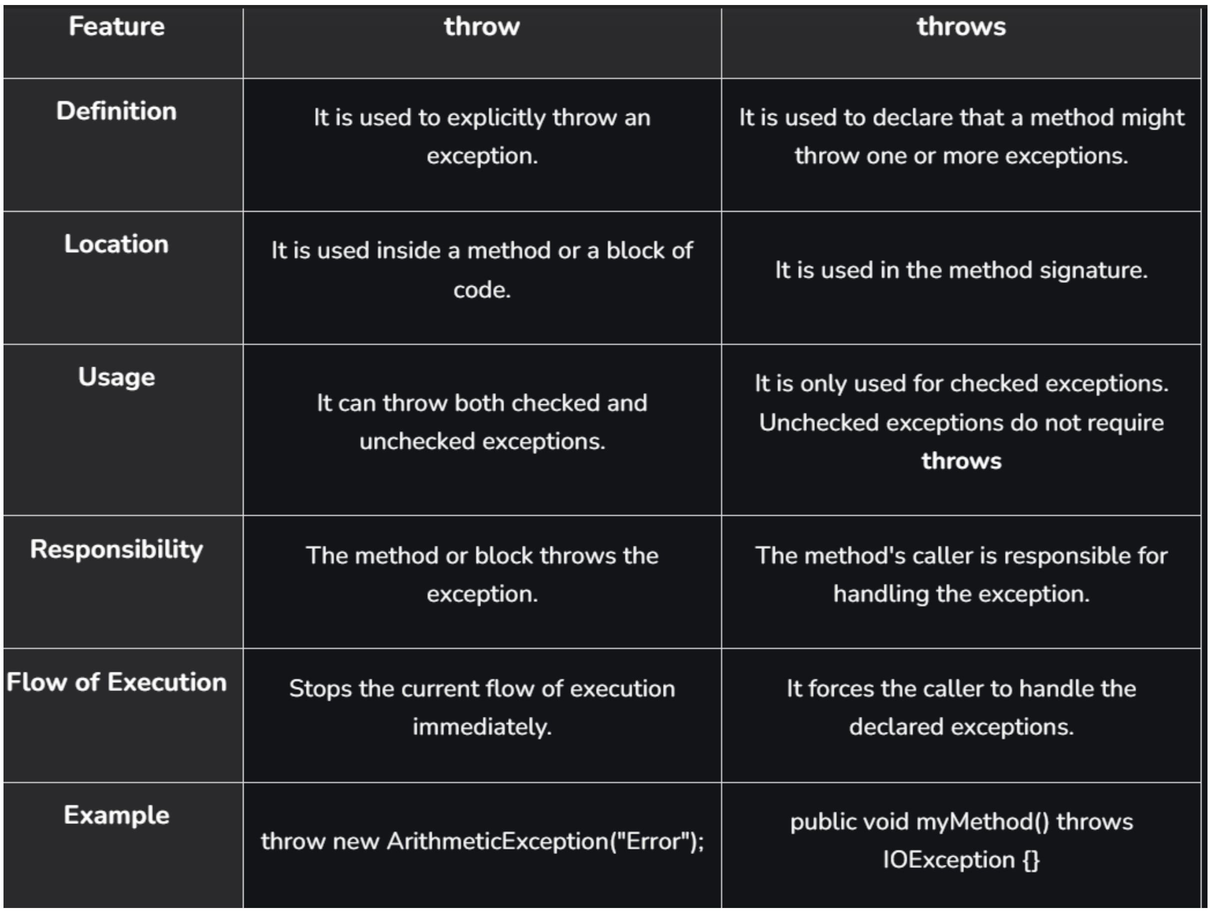
Example 3: Throwing an Exception with throws

```
// Demonstrating how to throw an exception
class ABC {

    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.");
        }
    }
}
```

Important Points:

- throws keyword is required only for checked exceptions and usage of the throws keyword for unchecked exceptions is meaningless.
- throws keyword is required only to convince the compiler and usage of the throws keyword does not prevent abnormal termination of the program.
- With the help of the throws keyword, we can provide information to the caller of the method about the exception.

Difference between throw and throws



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