Class 15 : Exception Handling Basics

by Pial Kanti Samadder

Overview of Errors and Exceptions

An exception is an *unwanted or unexpected event* that occurs during the execution of the program, that disrupts the flow of the program.

- For example, if a user is trying to divide an integer by 0 then it is an exception, as it is not possible mathematically.

There are various types of interruptions while executing any program like errors, exceptions, and bugs. These interruptions can be due to programming mistakes or due to system issues. Depending on the conditions they can be classified as errors and exceptions.

The Catch or Specify Requirement

Valid Java programming language code must honor the *Catch or Specify Requirement*.

- In Java, if code can throw a **checked exception**, you must do one of two things:
  - **Catch it** → Use a `try-catch` block to handle the exception.
  - **Specify it** → Declare in the method signature using `throws`.
- If you don't do either → the code **won't compile**.
- **Only checked exceptions** follow this rule.
  - Errors and runtime exceptions are **not** subject to it.

What is Exception

- An **Exception** is the occurrence of an event that can disrupt the normal flow of the program instructions.
- Exceptions can be caught and handled in order to keep the program working for the exceptional situation as well, instead of halting the program flow.
- If the exception is not handled, then it can result in the termination of the program.
- Exceptions can be used to indicate that an error has occurred in the program.

When an exception occurs, it creates an **exception object**. It holds information about the name and description of the exception and stores the program's state when the exception occurred.

The below image shows the flow of an exception:

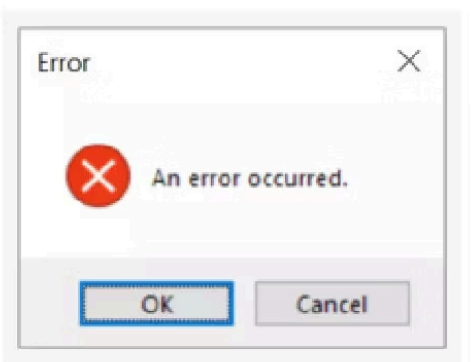
Types of Exceptions

There are *2 types* of Exceptions:

1. **Checked Exceptions**
   - Checked exceptions are those exceptions that are checked at compile time by the compiler.
   - The program will not compile if they are not handled.
   - These exceptions are child classes of the Exception class. IOException, ClassNotFoundException, InvocationTargetException, and SQL Exception are a few of the checked exceptions in Java.
2. **Unchecked Exceptions**
   - Unchecked exceptions are those exceptions that are checked at run time by JVM, as the compiler cannot check unchecked exceptions.
   - The programs with unchecked exceptions get compiled successfully but they give runtime errors if not handled.
   - These are child classes of Runtime Exception Class. ArithmeticException, NullPointerException, NumberFormatException, IndexOutOfBoundException are a few of the unchecked exceptions in Java.

What is Error

- An error is also an unwanted condition but it is caused due to lack of resources and indicates a serious problem.
- Errors are irrecoverable, they cannot be handled by the programmers.
- Errors are of unchecked type only.
- They can occur only at run time.
- In java, errors belong to `java.lang.error` class.
- *Eg*: `OutOfMemmoryError`.



Handling an error is out of the scope of a program. It can be handled **externally**.

Why Handle Java Exceptions?

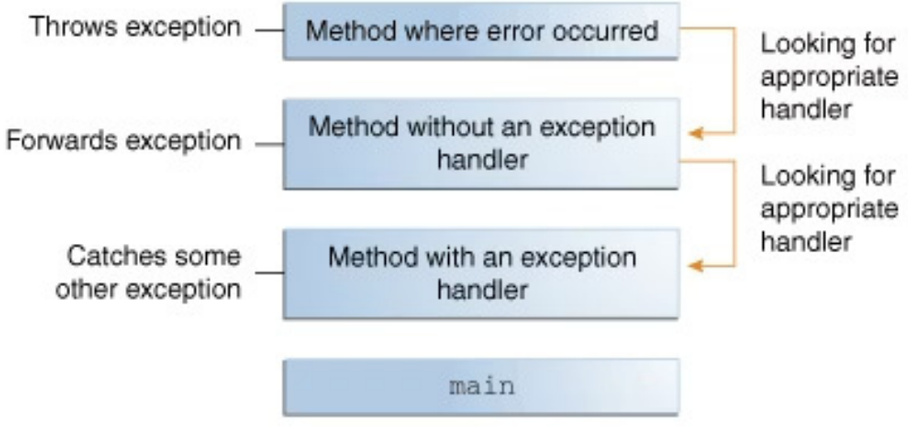
- To **prevent the program from crashing** when an error occurs.
- Exceptions can happen at **compile-time or runtime**, and even minor ones can stop execution.
- If exceptions are **not handled**, the program may terminate abruptly, leaving tasks unfinished.
- By handling exceptions, we can:
  - Keep the program **running smoothly**
  - **Maintain the flow** of execution
  - **Notify the user** about the problem instead of crashing

Example:

Without handling: program stops midway if an exception occurs.
With handling: program continues and executes remaining statements safely.

How Does JVM Handle an Exception?

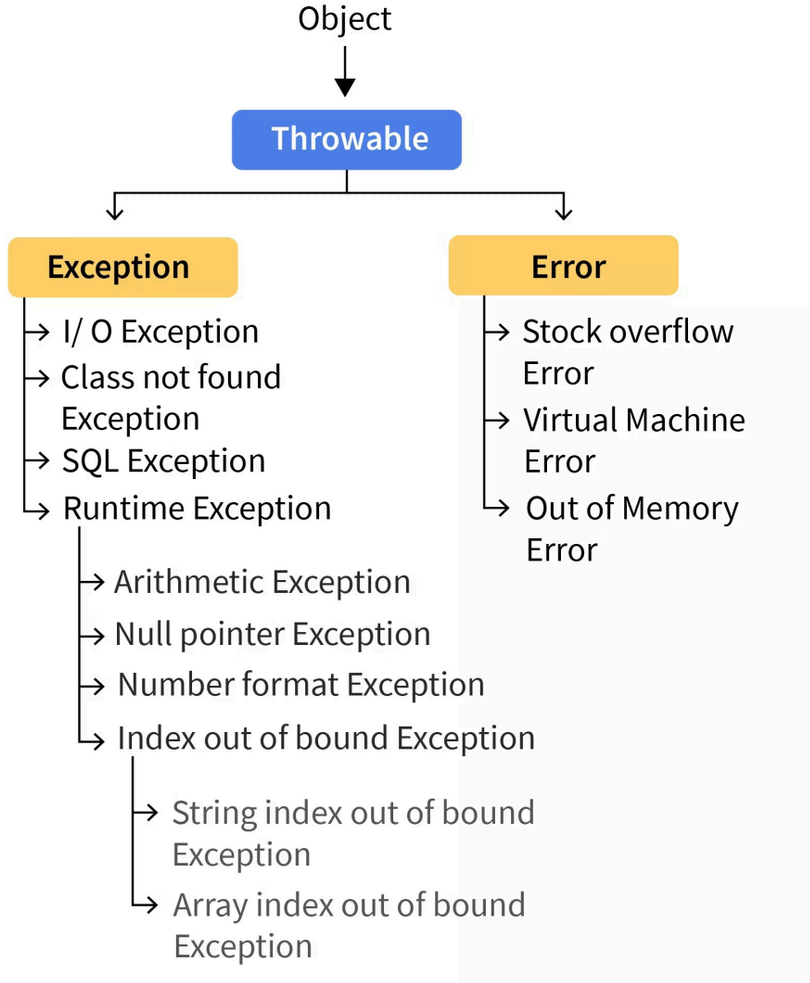
1. **Exception Creation**
   - When an error occurs, the method **creates an exception object** containing:
     - Type of exception
     - Description
     - State of the program
2. **Throwing the Exception**
   - The exception object is **handed to the JVM** → this is called **throwing an exception**.
3. **Searching for a Handler**
   - JVM looks through the **call stack** (the list of active method calls) to find a **matching exception handler**.
   - The handler is a block of code that knows how to process the exception.
4. **Handling or Default Termination**
   - If a handler is found → exception is **caught and handled**.
   - If no handler is found → JVM's **default handler** prints the exception and **terminates the program**.



Hierarchy of Java Exception Classes

All exceptions and errors are subclasses of class `Throwable`.

`Throwable` is the base/superclass of exception handling hierarchy in java. This class is further extended in different classes like exception, error, and so on.



How does a Programmer handle an Exception?

Customized exception handling in java is achieved using **five keywords**: `try`, `catch`, `throw`, `throws`, and `finally`. Here is how these keywords work in short.

- `Try` block contains the program statements that may raise an exception.
- `Catch` block catches the raised exception and handles it.
- `Throw` keyword is used to explicitly throw an exception.
- `Throws` keyword is used to declare an exception.
- `Finally` block contains statements that must be executed after the try block.

`try` block

- `try` block is used to execute doubtful statements which can throw exceptions.
- `try` block can have multiple statements.
- `try` block cannot be executed on itself, there has to be at least one catch block or finally block with a `try` block.
- When any exception occurs in a `try` block, the appropriate exception object will be redirected to the catch block, this catch block will handle the exception according to statements in it and continue the further execution.
- The control of execution goes from the `try` block to the catch block once an exception occurs.

Syntax

```
try
{
 //Doubtfull Statements.
}
```

`catch` block

- `catch` block is used to give a solution or alternative for an exception.
- `catch` block is used to handle the exception by declaring the type of exception within the parameter.
- The declared exception must be the parent class exception or the generated exception type in the exception class hierarchy or a user-defined exception.
- You can use multiple catch blocks with a single `try` block.

Syntax

```
try
{
 //Doubtful Statements
}
catch(Exception e)
{
```

Examples using try-catch blocks

Let's see a simple example of exception handling in java using a try-catch block.

```
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {10, 1, 2, 3, 5, 11};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

Output:

```
Something went wrong.
```

Multiple Catch Blocks

Sometimes, the code can throw more than one exception, and we can have more than one *catch* block handle each individually:

```
public int getPlayerScore(String playerFile) {
    try (Scanner contents = new Scanner(new File(playerFile))) {
        return Integer.parseInt(contents.nextLine());
    } catch (IOException e) {
        logger.warn("Player file wouldn't load!", e);
        return 0;
    } catch (NumberFormatException e) {
        logger.warn("Player file was corrupted!", e);
        return 0;
    }
}
```

Multiple catches give us the chance to handle each exception differently, should the need arise.

Also note here that we didn't catch `FileNotFoundException`, and that is because it *extends* `IOException`. Because we're catching *IOException*, Java will consider any of its subclasses also handled.

Let's say, though, that we need to treat *FileNotFoundException* differently from the more general *IOException*:

```
public int getPlayerScore(String playerFile) {
    try (Scanner contents = new Scanner(new File(playerFile)) ) {
        return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException e) {
        logger.warn("Player file not found!", e);
        return 0;
    } catch (IOException e) {
        logger.warn("Player file wouldn't load!", e);
        return 0;
    } catch (NumberFormatException e) {
        logger.warn("Player file was corrupted!", e);
        return 0;
    }
}
```

Java lets us handle subclass exceptions separately, **remember to place them higher in the list of catches.**

`finally` block

- `finally` block is associated with a `try`, `catch` block.
- **It is executed every time irrespective of exception is thrown or not**.
- `finally` block is used to execute important statements such as closing statement, release the resources, and release memory also.
- `finally` block can be used with **try** block with or without **catch** block.

```
public int getPlayerScore(String playerFile)
  throws FileNotFoundException {
    Scanner contents = null;
    try {
        contents = new Scanner(new File(playerFile));
        return Integer.parseInt(contents.nextLine());
    } finally {
        if (contents != null) {
            contents.close();
        }
    }
}
```

- Here, the *finally* block indicates what code we want Java to run regardless of what happens with trying to read the file.
- Even if a *FileNotFoundException* is thrown up the call stack, Java will call the contents of *finally* before doing that.
- We can also both handle the exception *and* make sure that our resources get closed:

`throws` keyword

- `throws` keyword in java is used in the signature of the method for **specifying the exceptions thrown by a Method**.

Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception.

For example, the `writeList` method doesn't catch the checked exceptions that can occur within it, the `writeList` method **must specify that it can throw these exceptions**. Here's the version of the `writeList` method that won't compile:

```
public void writeList() {
   PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
   for (int i = 0; i < SIZE; i++) {
      out.println("Value at: " + i + " = " + list.get(i));
   }
   out.close();
}
```

To specify that `writeList` can throw two exceptions, add a `throws` clause to the method declaration for the `writeList` method.

```
public void writeList() throws IOException, IndexOutOfBoundsException {
```

Since **unchecked exception in the `throws` clause is not mandatory**. You could just write the following.

```
// IndexOutOfBoundsException is an unchecked exception
public void writeList() throws IOException {
```

try-with-resources

Introduced in Java 7, try-with-resources simplifies the process of closing resources that implement the `AutoCloseable` or `Closeable` interface.

- Resources declared **within the try parentheses are automatically closed after the try block**.
- Can be combined with catch and/or finally blocks.

Example:

```
public int getPlayerScore(String playerFile) {
    try (Scanner contents = new Scanner(new File(playerFile))) {
      return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException e ) {
      logger.warn("File not found, resetting score.");
      return 0;
    }
}
```

Union `catch` Blocks

When we know that the way we handle errors is going to be the same, though, Java 7 introduced the ability to catch multiple exceptions in the same block:

```
public int getPlayerScore(String playerFile) {
    try (Scanner contents = new Scanner(new File(playerFile))) {
        return Integer.parseInt(contents.nextLine());
    } catch (IOException | NumberFormatException e) {
        logger.warn("Failed to load score!", e);
        return 0;
    }
}
```

Creating Custom Exceptions

Custom exceptions can make your code more readable and help differentiate your application's specific errors from standard Java exceptions. They are particularly useful when you need to add additional information to an exception or to clarify the purpose of an exception.

Implementation:

- Extend either `Exception` (for checked exceptions) or `RuntimeException` (for unchecked exceptions).
- Provide constructors that accept messages, cause, or both.

Example

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

    public MyCustomException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Throwing Exceptions in Java

- ## Throwing an Exception

Use the throw keyword to create and throw an exception object.

```
throw new IllegalArgumentException("Invalid age");
```

- ## Declaring Exceptions

For checked exceptions, the JVM forces you to use throws in the method signature.

```
public void readFile(String path) throws IOException {
    FileReader fr = new FileReader(path);
}
```

- ## Key Rules
  - `throw` → actually throws an exception object.
  - `throws` → declares that a method may throw exceptions.
  - Checked exceptions → must be caught or declared (Catch or Specify Requirement).
  - Runtime exceptions & errors → don’t need to be declared.

Exception Chaining

Exception chaining (also known as exception wrapping) is the process of catching an original exception and re-throwing a new exception that includes the original one. This technique is useful when you want to add additional context to an exception or translate lower-level exceptions to higher-level ones.

Implementation:

- Use exception constructors that accept another exception as a cause.
- The `getCause()` method can be used to retrieve the original exception.

Example

```
try {
    // Some code that might throw SQLException
} catch (SQLException e) {
    throw new MyCustomException("Database operation failed", e);
}
```

Best Practices in Exception Handling

1. **Catch Specific Exceptions**
   - Handle exact errors (e.g., `IOException`, `SQLException`) instead of catching a general `Exception`.
   - Improves readability and allows targeted fixes
2. **Avoid Empty Catch Blocks**
   - Empty catch blocks,or *exception swallowing*, can make debugging a nightmare since it hides errors.
   - Always **log or handle** them
3. **Use `finally` (or try-with-resources) for Cleanup**
   - `finally` block is executed regardless of whether an exception is thrown
   - Ensure resources are closed in the `finally` block
   - Release files, DB connections, network resources
4. **Throw Early, Catch Late**
   - Throw exceptions as soon as a problem is detected
   - Let low-level methods throw exceptions and have them handled at a higher level in the application
5. **Don't Catch `Throwable`, `Error`, or General Exceptions**
   - Catching `Throwable` or `Error` can lead to catching severe system errors that the application should not attempt to handle
   - Catching `RuntimeException` is often unnecessary and can mask bugs like `NullPointerExceptions`
   - Be specific and avoid catching very general exceptions or errors.
6. **Document Exceptions**
   - Use Javadoc (`@throws`) to list exceptions a method may throw
   - Helps other developers know what to handle