Exception handling in java

Team: Fast Trackers

Members:

Amrutha Muraleedharan Nair
Anirudh Gopalakrishnan
Blesson John Abraham
Jenson Mathew

Introduction

Exception handling in Java is the process of handling runtime errors and exceptional conditions that can occur in a Java program. When an exceptional condition occurs in a Java program, it generates an exception, which can be handled using the try-catch block.

Here's an example of how exception handling works in Java:

```
java
Copy code
try {
// code that may throw an exception
} catch (ExceptionType1 e1) {
// code to handle the exception of type ExceptionType1
} catch (ExceptionType2 e2) {
// code to handle the exception of type ExceptionType2
} finally {
// code that is executed regardless of whether an exception is thrown or not
}
```

In this example, the try block contains the code that may throw an exception. If an exception is thrown, the Java runtime searches for a matching catch block to handle the exception. Each catch block can handle exceptions of a specific type, and the block containing the matching type is executed. If no matching catch block is found, the exception is propagated up the call stack to the next higher-level try-catch block or to the default Java exception handler.

The finally block contains code that is executed regardless of whether an exception is thrown or not. This block is often used to clean up resources or perform other tasks that must be executed even if an exception occurs.

Some commonly used exceptions in Java include NullPointerException, IndexOutOfBoundsException, ArithmeticException, IllegalArgumentException, FileNotFoundException, IOException, ClassNotFoundException, and InterruptedException.

Types of exception handling with examples

In Java, there are two types of exceptions - checked exceptions and unchecked exceptions. Here are some examples of each type:

Checked Exceptions:
Checked exceptions are exceptions that are checked at compile-time. This means that the programmer is required to handle or declare these exceptions in their code. Some examples of checked exceptions are:

IOException - This exception is thrown when an input/output operation fails. For example, if a file cannot be found or opened, an IOException will be thrown.

java
Copy code
```
try {
// code that may throw an IOException
} catch (IOException e) {
// code to handle the IOException
}
```

ClassNotFoundException - This exception is thrown when a class is not found at runtime. For example, if a Java program tries to load a class that does not exist, a ClassNotFoundException will be thrown.
java
Copy code
```
try {
// code that may throw a ClassNotFoundException
} catch (ClassNotFoundException e) {
// code to handle the ClassNotFoundException
}
```

Unchecked Exceptions:
Unchecked exceptions are exceptions that are not checked at compile-time. This means that the programmer is not required to handle or declare these exceptions in their code. Some examples of unchecked exceptions are:

NullPointerException - This exception is thrown when a program attempts to use a null object reference. For example, if a program tries to call a method on a null object, a NullPointerException will be thrown.
java
Copy code
```
try {
// code that may throw a NullPointerException
} catch (NullPointerException e) {
// code to handle the NullPointerException
}
```

ArithmeticException - This exception is thrown when an arithmetic operation produces an error. For example, if a program tries to divide a number by zero, an ArithmeticException will be thrown.
java
Copy code

```
try {
// code that may throw an ArithmeticException
} catch (ArithmeticException e) {
// code to handle the ArithmeticException
}
```

It's important to note that while checked exceptions are required to be handled or declared, it's not always necessary to handle unchecked exceptions. However, it's generally good practice to handle all exceptions in your code to prevent unexpected behavior or crashes

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.

throw *Instance*
Example:
throw new ArithmeticException("/ by zero");

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**. For example Exception is a sub-class of Throwable and user defined exceptions typically extend Exception class. 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 Custom Exceptions.

Java provides us the facility to create our own exceptions which are basically derived classes of Exception. Creating our own Exception is known as a custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user needs. In simple words, we can say that a User-Defined Exception or custom exception is creating your own exception class and throwing that exception using the 'throw' keyword.

Why use custom exceptions?

Some of the reasons are:

- To catch and provide specific treatment to a subset of existing Java exceptions.
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem

In order to create a custom exception, we need to extend the Exception class that belongs to **java.lang package.**

In Java, custom exceptions can be created by extending the Exception class or any of its subclasses. Custom exceptions can be used to provide more descriptive error messages, handle specific errors, and improve the readability and maintainability of the code.

Here are some examples of custom exceptions in Java:

Example 1: Custom exception extending Exception class

java
Copy code
```
public class InvalidAgeException extends Exception {
public InvalidAgeException(String errorMessage) {
super(errorMessage);
}
}
```
In the above example, a custom exception called InvalidAgeException is created by extending the Exception class. The constructor of the custom exception takes an error message as input, which is passed to the constructor of the Exception class using the super() keyword.

Example usage:

java
Copy code
```
public void validateAge(int age) throws InvalidAgeException {
if (age < 0 || age > 120) {
throw new InvalidAgeException("Invalid age: " + age);
}
}
```
In the above example, a method called validateAge() takes an integer age as input and throws an InvalidAgeException if the age is outside the valid range of 0 to 120.

Example 2: Custom exception extending RuntimeException class

java
Copy code
```
public class FileFormatException extends RuntimeException {
public FileFormatException(String errorMessage) {
super(errorMessage);
}
}
```
In the above example, a custom exception called FileFormatException is created by extending the RuntimeException class. The constructor of the custom exception takes an error message as input, which is passed to the constructor of the RuntimeException class using the super() keyword.

Example usage:

typescript
Copy code
```
public void readFile(String fileName) {
```

```
if (!fileName.endsWith(".txt")) {
throw new FileFormatException("Invalid file format: " + fileName);
}
// read the file
}
```

In the above example, a method called readFile() takes a string fileName as input and throws a FileFormatException if the file format is not .txt. The FileFormatException is an unchecked exception, which means it does not need to be declared in the method signature using the throws keyword