Java Exception Handling

In Java, an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Exception handling is the mechanism to handle such exceptional conditions gracefully, preventing abrupt termination of the program.

Types of Exceptions:

1. Checked Exceptions:

These are exceptional conditions that a well-written application should anticipate and recover from. Examples include **IOException, SQLException**, etc. Checked exceptions must be caught or declared using a `throws` clause in the method signature.

2. Unchecked Exceptions (Runtime Exceptions):

These are exceptional conditions that are not checked at compile time. Examples include **NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException,** etc. Unchecked exceptions do not need to be explicitly caught or declared.

Syntax:

```
try {
        // Code that may throw an exception
        }
catch (ExceptionType1 e1) {
        // Handler for ExceptionType1
        }
catch (ExceptionType2 e2) {
        // Handler for ExceptionType2
        }
finally {
        // Code to be executed regardless of whether an exception
occurred or not
        }
```

try: Encloses the code that may throw an exception.

catch: Catches and handles exceptions of specified types.

finally: Contains code that is always executed, regardless of whether an exception occurs or not.

Example:

```
try {
        int result = 10 / 0; // Division by zero
        }
catch (ArithmeticException e) {
        System.out.println("Error: Division by zero");
        }
```

Using **finally** block:

```
FileInputStream file = null;
try {
        file = new FileInputStream("example.txt");
        // Code to read from file
}catch (IOException e) {
        System.out.println("Error reading file");
}finally {
        if (file != null) {
            try {
                file.close(); // Ensure file is closed
            }catch (IOException e) {
                System.out.println("Error closing file");
            }
        }
}
```

Key Points:

- It's good practice to handle specific exceptions rather than catching generic **`Exception`.**
- Exceptions should be handled appropriately to provide meaningful error messages and maintain program stability.
- The **`finally`** block is used to perform cleanup actions, such as closing resources, and is executed whether an exception occurs or not.
- Java provides a hierarchical structure of exceptions, with **`Throwable`** at the top. Both **`Exception`** and **`Error`** classes are subclasses of **`Throwable`.**
- Custom exceptions can be created by extending the **`Exception`** class or one of its subclasses.

Java exception handling is crucial for writing robust and reliable applications. Proper handling of exceptions enhances the resilience and maintainability of Java programs.

Throws key word

Certainly! Let's delve into the `throws` keyword in Java exception handling:

Java Exception Handling

In Java, the `**throws**` keyword is used in method signatures to indicate that a method might throw certain exceptions during its execution. When a method is declared with `**throws**`, it means that the method does not handle the exceptions itself, but rather it declares that it may propagate the exceptions to its caller.

Syntax:

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

Example:

```
public void openFile() throws FileNotFoundException {
    FileInputStream file = new FileInputStream("example.txt");
    // Code to read from file
}
```

In the above example, the `openFile()` method declares that it may throw a `FileNotFoundException` if the specified file is not found.

When to Use throws:

1. **Propagating Checked Exceptions**:

When a method encounters a checked exception that it cannot handle, it can declare the exception using `throws` in its method signature, allowing the caller method to handle it.

2. Delegating Exception Handling:

Sometimes, a method may not have enough context to handle an exception effectively. In such cases, it can delegate the responsibility of handling exceptions to the caller method by using `throws`.

Considerations:

1**. Checked vs. Unchecked Exceptions:**

The `throws` keyword is typically used with checked exceptions because unchecked exceptions do not need to be declared in method signatures.

2. **Handling vs. Propagating Exceptions:**

Methods can either handle exceptions using `try-catch` blocks or propagate them using `throws`, depending on whether they can effectively handle the exception or not.

Example with `throws`:

```
public void processFile() throws IOException {
    try {
        openFile();
    } catch (FileNotFoundException e) {
        System.out.println("File not found: " + e.getMessage());
    }
}
```

In the above example, the `processFile()` method calls the `openFile()` method, which may throw a `FileNotFoundException`. Since `processFile()` cannot handle this exception itself, it declares that it may propagate an `IOException`.

Key Points:

- The `throws` keyword is used to declare exceptions that a method may throw during its execution.
- It is used in method signatures and indicates that the method does not handle the exceptions itself but may propagate them to its caller.
- Methods can declare multiple exceptions separated by commas using `throws`.
- The caller method of a method with `throws` must either handle the declared exceptions using `try-catch` blocks or propagate them further.

Java's `throws` keyword allows for a clear declaration of exception propagation, aiding in designing robust and maintainable code.

Custom Exceptions in Java

In Java, custom exceptions are user-defined exception classes that extend either the `Exception` class or one of its subclasses. They are useful for representing application-specific error conditions that are not adequately captured by the built-in exception types.

Creating Custom Exceptions:

To create a custom exception, follow these steps:

1. Define a new class:

   Create a new class that extends `Exception` (or another appropriate subclass like `RuntimeException`).

2. Provide Constructors:

   Define constructors to initialize the exception object. You may want to provide constructors with different parameters to customize error messages or capture additional information.

3. Optionally Override Methods:

   You can optionally override methods from the superclass to add custom behavior.

Example:

```
public class CustomException extends Exception {

    public CustomException() {
        super();
    }

    public CustomException(String message) {
        super(message);
    }

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

```
    }
}
```

In this example, `CustomException` is a custom exception class that extends `Exception` and provides constructors to initialize the exception object with custom error messages and causes.

Using Custom Exceptions:

Once you have defined a custom exception class, you can use it in your application like any other exception type. Here's how you might use the `CustomException` class:

```
public class MyClass {

    public void someMethod() throws CustomException {
        // Some logic that may throw CustomException
        throw new CustomException("Custom error message");
    }

    public static void main(String[] args) {
        MyClass obj = new MyClass();
        try {
            obj.someMethod();
        } catch (CustomException e) {
            System.out.println("Custom exception caught: " + e.getMessage());
        }
    }
}
```

In this example, the `someMethod()` method declares that it may throw a `CustomException`. When an instance of `MyClass` invokes this method, it handles the exception using a `try-catch` block.

Key Points:

- Custom exceptions allow developers to represent application-specific error conditions.
- They are created by defining a new class that extends `Exception` or its subclasses.
- Custom exceptions typically provide constructors to initialize the exception object with custom error messages or causes.
- Custom exceptions can be thrown and caught like any other exception type in Java.

Best Practices:

1. **Be Descriptive:** Provide meaningful names for custom exceptions that clearly describe the error condition.

2**. Include Useful Information:** Customize constructors to include relevant information that aids in debugging and error handling.

3. **Follow Exception Handling Best Practices:** Handle custom exceptions appropriately in your application code, either by catching them or propagating them to higher levels.

By creating custom exceptions, developers can improve the clarity and maintainability of their code by explicitly handling application-specific error conditions.