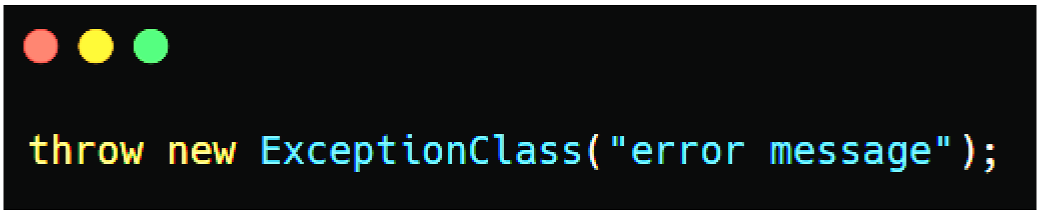
11.5-Exception throw keyword

Introduction to throw Keyword

In Java, when an exception occurs, it is typically handled using **try-catch** blocks. However, we can also manually generate exceptions using the **throw** keyword. The throw keyword is used to explicitly throw an exception during the execution of a program.

This is helpful when you want to create and manage your own custom exceptions or handle specific error conditions in a controlled manner. For example, you might throw an exception when user input is invalid, when a server fails to respond, or when certain logical conditions in your code are met.

- **Syntax of throw keyword**:



```java
throw new ExceptionClass("error message");
```

Here, an instance of the exception class is created with an error message, and the throw keyword passes this exception to the appropriate catch block.

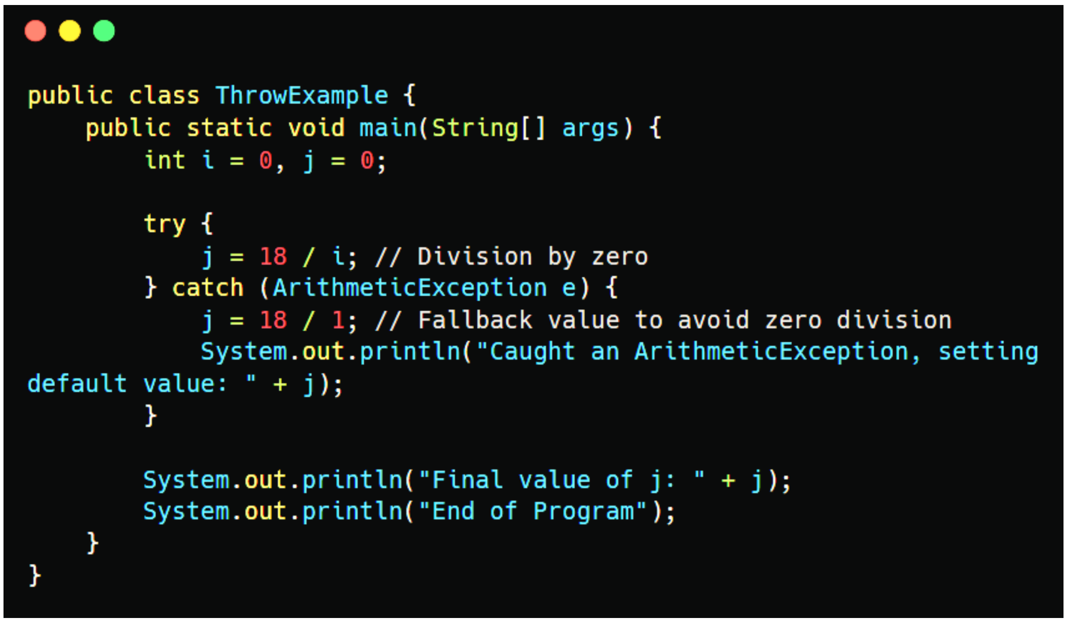
When to Use the throw Keyword

You can use the throw keyword to throw both **checked** and **unchecked** exceptions. This is useful when:

- You want to define your own conditions and throw an exception explicitly.
- You want to generate an exception based on specific logic (e.g., dividing by zero or invalid data).

For example, if you're dividing two numbers, you can throw an ArithmeticException if the denominator is zero.

Example 1: Handling ArithmeticException

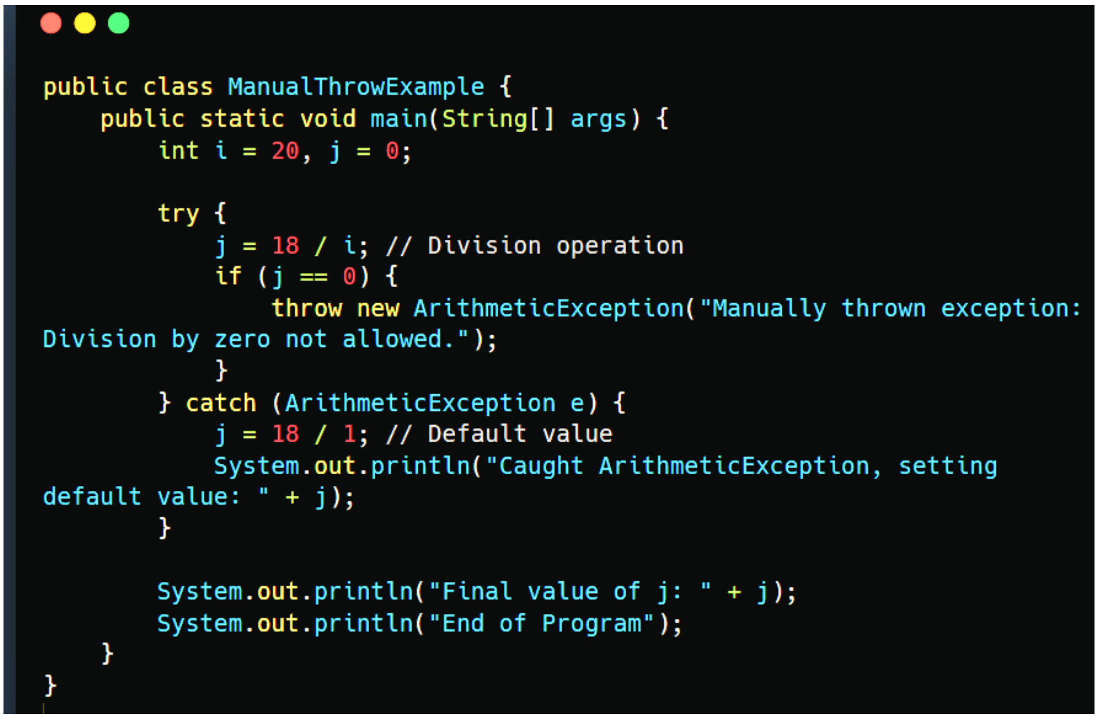


```java
public class ThrowExample {
    public static void main(String[] args) {
        int i = 0, j = 0;

        try {
            j = 18 / i; // Division by zero
        } catch (ArithmeticException e) {
            j = 18 / 1; // Fallback value to avoid zero division
            System.out.println("Caught an ArithmeticException, setting
default value: " + j);
        }

        System.out.println("Final value of j: " + j);
        System.out.println("End of Program");
    }
}
```

Explanation:

- In the above code, i is set to 0. When the program tries to divide by zero, an **ArithmeticException** is thrown.
- The catch block catches the exception and sets a default value of 18.
- The program continues to execute after handling the exception.

Example 2: Manually Throwing an Exception



```java
public class ManualThrowExample {
    public static void main(String[] args) {
        int i = 20, j = 0;

        try {
            j = 18 / i; // Division operation
            if (j == 0) {
                throw new ArithmeticException("Manually thrown exception:
Division by zero not allowed.");
            }
        } catch (ArithmeticException e) {
            j = 18 / 1; // Default value
            System.out.println("Caught ArithmeticException, setting
default value: " + j);
        }

        System.out.println("Final value of j: " + j);
        System.out.println("End of Program");
    }
}
```

Explanation:

- In this example, a condition is added that manually throws an **ArithmeticException** when j equals 0.
- Even though no actual division by zero occurs in the program, the exception is thrown explicitly, allowing custom error handling.

This technique is commonly used in scenarios like database connections, where **fallback strategies** are needed when a primary resource (like a database) fails.

Key Points About the throw Keyword:

- **Use it for Custom Exceptions**: You can create and throw custom exceptions based on your application's needs.
- **Explicit Handling**: It allows explicit error handling at specific points in your code, offering more control over how and when exceptions are triggered.
- **Default Fallbacks**: You can throw exceptions and handle them gracefully by providing fallback solutions or retry mechanisms, ensuring smooth execution flow.