DEV GENESIS

VIDEO 7: EXCEPTION HANDLING

Mastering error handling for robust applications

WHAT WE'LL COVER

- What are exceptions?
- Common runtime issues
- The try-catch-finally blocks
- Exception hierarchy
- Handling specific exceptions
- Best practices for exception handling
- Creating your own exceptions

WHAT ARE EXCEPTIONS?

Exceptions are **unexpected events** that disrupt the normal flow of a program.
They represent **error conditions** that occur during program execution.
Without proper handling, exceptions cause programs to **terminate abruptly**.

> Example of an unhandled exception:
>
> ```
> int[] array = new int[5];
> array[10] = 50;  // Accessing an index outside the array bounds
>
> // Output: ArrayIndexOutOfBoundsException: Index 10 out of bounds for
> length 5
> ```

WHY EXCEPTION HANDLING?

- Makes programs **robust** by handling unexpected situations
- Provides a mechanism to **separate error-handling code** from regular code
- Allows programs to **continue execution** despite errors
- Provides **meaningful feedback** to users instead of cryptic error messages
- Helps in **debugging** by providing information about what went wrong

COMMON RUNTIME EXCEPTIONS

Exception	Description	Example

NullPointerException	Accessing a null reference	`String s = null;`
		`s.length();`

ArithmeticException	Arithmetic error (divide by zero)	`int result = 10 / 0;`

ArrayIndexOutOfBoundsException	Invalid array index	`int[] a = new int[5];`
		`a[10];`

NumberFormatException	Failed to convert string to number	`Integer.parseInt("abc");`

ClassCastException	Invalid cast operation	`Object x = "string";`
		`Integer i = (Integer)x;`

THE TRY-CATCH BLOCK

Basic syntax for handling exceptions:

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
}
```

try block
Contains code that might throw an exception

catch block
Executes only if an exception occurs in the try block
Has access to the exception object

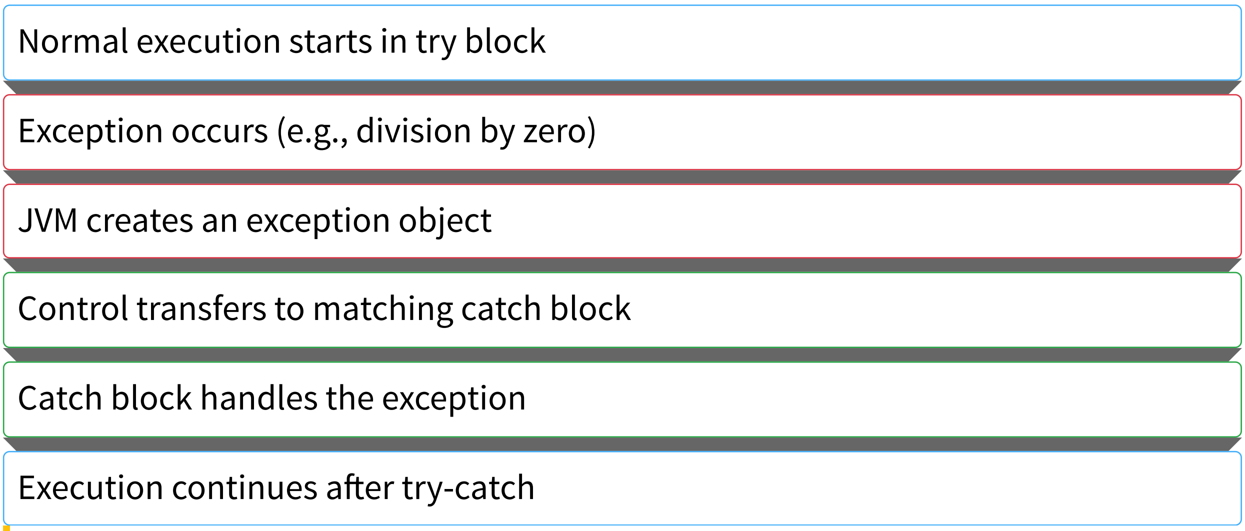
EXAMPLE: DIVISION WITH EXCEPTION HANDLING

```
public class DivisionExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 0;

        // Without exception handling
        // int result = a / b;  // This will cause the program to crash

        // With exception handling
        try {
            System.out.println("Attempting to divide " + a + " by " +
b);
            int result = a / b;
            System.out.println("Result: " + result);  // Won't execute
if b is 0
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero!");
            System.out.println("Exception message: " + e.getMessage());
        }

        System.out.println("Program continues execution...");
    }
}
```

Output:

```
Attempting to divide 10 by 0
Error: Cannot divide by zero!
Exception message: / by zero
Program continues execution...
```

EXCEPTION FLOW



Normal execution starts in try block

Exception occurs (e.g., division by zero)

JVM creates an exception object

Control transfers to matching catch block

Catch block handles the exception

Execution continues after try-catch

> If no exception occurs, the catch block is skipped entirely.

MULTIPLE CATCH BLOCKS

You can handle different types of exceptions differently:

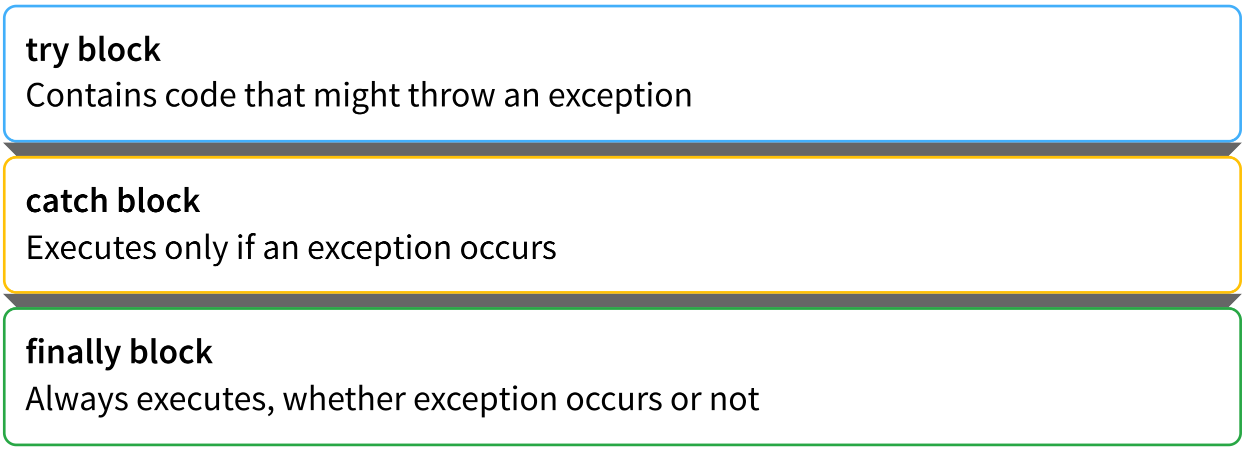
```java
try {
    int[] numbers = {1, 2, 3};
    int result = numbers[5] / 0;  // Potential for multiple exceptions
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index error: " + e.getMessage());
} catch (ArithmeticException e) {
    System.out.println("Arithmetic error: " + e.getMessage());
} catch (Exception e) {  // Catches any other exceptions
    System.out.println("Something else went wrong: " + e.getMessage());
}
```

> Order matters! Place more specific exception types before more general ones.

THE FINALLY BLOCK

The `finally` block contains code that always executes, regardless of whether an exception occurs:

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
    // Code that always executes
}
```



try block
Contains code that might throw an exception

catch block
Executes only if an exception occurs

finally block
Always executes, whether exception occurs or not

EXAMPLE WITH FINALLY BLOCK

```java
public class ResourceExample {
    public static void main(String[] args) {
        // Simulating a resource that needs to be closed
        System.out.println("Opening a resource...");

        try {
            System.out.println("Working with the resource");
            // Simulating an error
            int result = 10 / 0;
            System.out.println("This line won't execute if there's an
exception");
        } catch (ArithmeticException e) {
            System.out.println("Error occurred: " + e.getMessage());
        } finally {
            // This will always execute, ensuring the resource is closed
            System.out.println("Closing the resource");
        }

        System.out.println("Program continues...");
    }
}
```

Output:

```
Opening a resource...
Working with the resource
Error occurred: / by zero
Closing the resource
Program continues...
```

COMMON USES OF FINALLY

- Closing database connections
- Closing file streams
- Releasing network resources
- Releasing locks
- Cleaning up temporary resources

> The `finally` block is ideal for cleanup operations that must happen regardless of success or failure.

> In modern Java, many resource management tasks are better handled with try-with-resources (introduced in Java 7).

TRY-WITH-RESOURCES

A cleaner way to handle resources that need to be closed:

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TryWithResourcesExample {
    public static void main(String[] args) {
        // The resource will be automatically closed when the try block
exits
        try (BufferedReader reader = new BufferedReader(
                new FileReader("file.txt"))) {
            String line = reader.readLine();
            System.out.println(line);
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
        // No need for finally block to close the reader
    }
}
```

> Resources must implement the `AutoCloseable` interface to be used with try-with-resources.

EXCEPTION HIERARCHY

- **Throwable** - Base class for all errors and exceptions
    - **Error** - Severe errors that programs usually shouldn't catch
        - OutOfMemoryError
        - StackOverflowError
    - **Exception** - Base class for exceptions that programs can catch
        - Checked exceptions (must be caught or declared)
            - IOException
            - SQLException
        - RuntimeException (unchecked exceptions)
            - NullPointerException
            - ArithmeticException
            - ArrayIndexOutOfBoundsException

CHECKED VS. UNCHECKED EXCEPTIONS

CHECKED EXCEPTIONS

- Must be caught or declared in method signature
- Represent recoverable conditions
- Example: IOException, SQLException
- Compiler enforces handling

```
// Must either catch or declare it
void readFile() throws IOException {
    // Code that might throw IOException
}
```

UNCHECKED EXCEPTIONS

- Don't need to be caught or declared
- Usually represent programming errors
- Subclasses of RuntimeException
- Compiler doesn't enforce handling

```
// No need to declare it
void divide(int a, int b) {
    // Might throw ArithmeticException
    int result = a / b;
}
```

GETTING INFORMATION FROM EXCEPTIONS

The Exception object provides several useful methods:

```java
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    // Get the exception message
    System.out.println("Message: " + e.getMessage());

    // Get the class name of the exception
    System.out.println("Type: " + e.getClass().getName());

    // Print the stack trace (most detailed)
    e.printStackTrace();

    // Get the cause of this exception (if it was wrapped)
    Throwable cause = e.getCause();
}
```

> The stack trace shows the exact line where the exception occurred and the call stack at that point.

CREATING CUSTOM EXCEPTIONS - PART 1

You can create your own exception classes:

```
// Custom checked exception
public class InsufficientFundsException extends Exception {
    private double amount;

    public InsufficientFundsException(double amount) {
        super("Insufficient funds: You need " + amount + " more
dollars");
        this.amount = amount;
    }

    public double getAmount() {
        return amount;
    }
}
```

CREATING CUSTOM EXCEPTIONS - PART 2

Using the custom exception:

```
public class BankAccount {
    private double balance;

    public void withdraw(double amount) throws
InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException(amount - balance);
        }
        balance -= amount;
    }

    // Example usage
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        // Account has 0 balance

        try {
            account.withdraw(100);
        } catch (InsufficientFundsException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

THROWING EXCEPTIONS - PART 1

You can explicitly throw exceptions using the `throw` keyword:

```
public void verifyAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("Age cannot be negative");
    }

    if (age < 18) {
        throw new RuntimeException("Must be 18 or older");
    }

    System.out.println("Age verified successfully");
}
```

THROWING EXCEPTIONS - PART 2

When you throw a checked exception, you must declare it or catch it:

```java
// Method declares the checked exception
public void readFile(String filename) throws IOException {
    if (!fileExists(filename)) {
        throw new IOException("File not found: " + filename);
    }
    // Rest of the code...
}

// Catching the declared exception
public void processFile(String filename) {
    try {
        readFile(filename);
    } catch (IOException e) {
        System.out.println("Could not process file: " + e.getMessage());
    }
}
```

EXCEPTION HANDLING BEST PRACTICES

- Catch only exceptions you can handle
- Don't catch exceptions and do nothing (avoid empty catch blocks)
- Use specific exception types rather than catching all exceptions
- Provide meaningful error messages
- Clean up resources properly using finally or try-with-resources
- Don't use exceptions for normal flow control
- Log exceptions for debugging purposes
- Consider wrapping low-level exceptions in higher-level ones

DIVISION CALCULATOR EXAMPLE - PART 1

```
import java.util.Scanner;

public class DivisionCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            // Get input from user
            System.out.print("Enter the numerator: ");
            int numerator = Integer.parseInt(scanner.nextLine());

            System.out.print("Enter the denominator: ");
            int denominator = Integer.parseInt(scanner.nextLine());

            // Perform division
            double result = divide(numerator, denominator);

            // Display result
            System.out.println(numerator + " / " + denominator + " = " +
result);
        } catch (NumberFormatException e) {
            System.out.println("Error: Please enter valid integer
numbers.");
        }
```

DIVISION CALCULATOR EXAMPLE - PART 2

```java
        catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Unexpected error: " + e.getMessage());
        } finally {
            scanner.close();
        }
    }

    public static double divide(int numerator, int denominator) {
        if (denominator == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }
        return (double) numerator / denominator;
    }
}
```

> This example handles both invalid input (NumberFormatException) and divide-by-zero (ArithmeticException).

YOUR CODING EXERCISE

🧑‍💻 Create a Temperature Converter program with error handling:

1. Create a new Java class named `TemperatureConverter`
2. Ask the user to input a temperature in Celsius
3. Convert it to Fahrenheit using the formula: F = C × 9/5 + 32
4. Add exception handling to handle invalid inputs (non-numeric values)
5. Add validation to check if the temperature is within a reasonable range (e.g., -273.15°C to 5000°C)
6. Create a custom exception called `InvalidTemperatureException`
7. Allow the user to continue entering temperatures until they choose to quit

SOLUTION OUTLINE - PART 1

```java
import java.util.Scanner;

// Custom exception for invalid temperatures
class InvalidTemperatureException extends Exception {
    public InvalidTemperatureException(String message) {
        super(message);
    }
}

public class TemperatureConverter {
    // Constants for temperature validation
    private static final double MIN_CELSIUS = -273.15; // Absolute zero
    private static final double MAX_CELSIUS = 5000;    // Very high
temperature

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        boolean continueProgram = true;
```

SOLUTION OUTLINE - PART 2

```java
        while (continueProgram) {
            try {
                // Get user input
                System.out.print("Enter a temperature in Celsius (or 'q' to quit): ");
                String input = scanner.nextLine();

                // Check if user wants to quit
                if (input.equalsIgnoreCase("q")) {
                    continueProgram = false;
                    continue;
                }

                // Parse the input to a double
                double celsius = Double.parseDouble(input);

                // Validate temperature range
                validateTemperature(celsius);

                // Convert to Fahrenheit
                double fahrenheit = celsiusToFahrenheit(celsius);

                // Display the result
                System.out.printf("%.2f°C = %.2f°F%n", celsius, fahrenheit);

            } catch (NumberFormatException e) {
                System.out.println("Error: Please enter a valid number.");
            }
```

SOLUTION OUTLINE - PART 3

```
            catch (InvalidTemperatureException e) {
                System.out.println("Error: " + e.getMessage());
            } catch (Exception e) {
                System.out.println("Unexpected error: " +
e.getMessage());
            }

            System.out.println(); // Empty line for readability
        }

        System.out.println("Thank you for using the Temperature
Converter!");
        scanner.close();
    }

    // Method to validate temperature
    private static void validateTemperature(double celsius)
                                      throws InvalidTemperatureException
{
        if (celsius < MIN_CELSIUS) {
            throw new InvalidTemperatureException(
                "Temperature below absolute zero (" + MIN_CELSIUS +
"°C)");
        }
        if (celsius > MAX_CELSIUS) {
            throw new InvalidTemperatureException(
                "Temperature too high (maximum " + MAX_CELSIUS + "°C)");
        }
    }

    // Method to convert Celsius to Fahrenheit
    private static double celsiusToFahrenheit(double celsius) {
        return celsius * 9 / 5 + 32;
    }
}
```