Java Exceptions and Exception Handling

Introduction to Exceptions

An **exception** is an unexpected event that occurs during program execution and disrupts the normal flow of the program. Java provides a robust exception handling mechanism to handle runtime errors and maintain the normal flow of the application.

Key Benefits of Exception Handling:

- Separates error handling code from normal code
- Provides meaningful error messages
- Maintains program flow even when errors occur
- Makes debugging easier

Exception Hierarchy

Java exceptions follow a hierarchical structure:

```
java.lang.Object
    └── java.lang.Throwable
        ├── java.lang.Error
        └── java.lang.Exception
            ├── RuntimeException (Unchecked)
            └── Other Exceptions (Checked)
```

Throwable: Root class for all exceptions and errors **Error**: Serious problems that applications shouldn't try to handle **Exception**: Conditions that applications might want to catch

---

Types of Exceptions

1. Checked Exceptions

- Exceptions that are checked at compile time. Must be handled or declared.

Examples:

- IOException
- SQLException
- ClassNotFoundException

2. Unchecked Exceptions (Runtime Exceptions)

Exceptions that occur at runtime and are not checked at compile time.

Examples:

- NullPointerException
- ArrayIndexOutOfBoundsException
- IllegalArgumentException

3. Errors

Serious problems that applications shouldn't handle.

Examples:

- OutOfMemoryError
- StackOverflowError

Exception Handling Keywords

Java provides five keywords for exception handling:

1. **try**: Contains code that might throw an exception
2. **catch**: Handles specific exceptions
3. **finally**: Executes regardless of exception occurrence
4. **throw**: Manually throws an exception
5. **throws**: Declares exceptions that a method might throw

Try-Catch Block

Basic Syntax

```
try {
   // Code that might throw an exception
} catch (ExceptionType e) {
   // Exception handling code
}
```

Example 1: Handling ArithmeticException

```
public class BasicExceptionExample {
   public static void main(String[] args) {
      try {
         int a = 10;
         int b = 0;
         int result = a / b; // This will throw ArithmeticException
         System.out.println("Result: " + result);
      } catch (ArithmeticException e) {
         System.out.println("Error: Cannot divide by zero!");
         System.out.println("Exception message: " + e.getMessage());
      }
      System.out.println("Program continues...");
   }
}
```

Output:

```
Error: Cannot divide by zero!
Exception message: / by zero
Program continues...
```

Example 2: Handling ArrayIndexOutOfBoundsException

```
public class ArrayExceptionExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3, 4, 5};
            System.out.println("Accessing element at index 10: " + numbers[10]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Array index is out of bounds!");
            System.out.println("Exception details: " + e.toString());
        }
        System.out.println("Program execution completed.");
    }
}
```

Output:

```
Error: Array index is out of bounds!
Exception details: java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 5
Program execution completed.
```

Multiple Catch Blocks

You can handle different types of exceptions with multiple catch blocks.

Example: Multiple Catch Blocks

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            String str = null;
            int length = str.length(); // NullPointerException
            int[] arr = new int[5];
            arr[10] = 25; // ArrayIndexOutOfBoundsException
            int result = 10 / 0; // ArithmeticException
        } catch (NullPointerException e) {
            System.out.println("Null Pointer Exception caught: " +
e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index Exception caught: " +
e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic Exception caught: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("General Exception caught: " + e.getMessage());
        }
        System.out.println("Program continues after exception handling.");
    }
}
```

Output:

```
Null Pointer Exception caught: null
Program continues after exception handling.
```

Multi-Catch Block (Java 7+)

```
public class MultiCatchExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            int result = numbers[5] / 0;
        } catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
            System.out.println("Mathematical or Array error occurred: " +
e.getClass().getSimpleName());
            System.out.println("Message: " + e.getMessage());
        }
    }
}
```

Output:

```
Mathematical or Array error occurred: ArrayIndexOutOfBoundsException
Message: Index 5 out of bounds for length 3
```

Finally Block

- The finally block executes regardless of whether an exception occurs or not.

Example: Finally Block

```
public class FinallyExample {
   public static void main(String[] args) {
      try {
         System.out.println("Inside try block");
         int result = 10 / 0;
      } catch (ArithmeticException e) {
         System.out.println("Inside catch block: " + e.getMessage());
      } finally {
         System.out.println("Inside finally block - This always executes");
      }

      System.out.println("After try-catch-finally");
   }
}
```

Output:

```
Inside try block
Inside catch block: / by zero
Inside finally block - This always executes
After try-catch-finally
```

Try-with-Resources (Java 7+)

- Automatically closes resources that implement AutoCloseable.

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

public class TryWithResourcesExample {
   public static void main(String[] args) {
      try (FileReader file = new FileReader("nonexistent.txt")) {
```

```
            // Code to read file
            System.out.println("File opened successfully");
        } catch (IOException e) {
            System.out.println("File operation failed: " + e.getMessage());
        }
        // FileReader is automatically closed here
        System.out.println("Resources automatically closed");
    }
}
```

Output:

```
File operation failed: nonexistent.txt (The system cannot find the file specified)

Resources automatically closed
```

Throw and Throws

Throw Statement

- Used to manually throw an exception.

```
public class ThrowExample {
    public static void validateAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative: " +
age);
        }
        if (age > 150) {
            throw new IllegalArgumentException("Age seems unrealistic: " + age);
        }
        System.out.println("Valid age: " + age);
    }

    public static void main(String[] args) {
        try {
            validateAge(25);
            validateAge(-5);
        } catch (IllegalArgumentException e) {
            System.out.println("Validation Error: " + e.getMessage());
```

```
        }
    }
}
```

Output:

```
Valid age: 25
Validation Error: Age cannot be negative: -5
```

Throws Declaration

Used to declare that a method might throw specific exceptions.

```
import java.io.IOException;
public class ThrowsExample {

    // Method declares it might throw IOException
    public static void readFile(String filename) throws IOException {
        if (filename == null) {
            throw new IOException("Filename cannot be null");
        }
        System.out.println("Reading file: " + filename);
    }

    public static void main(String[] args) {
        try {
            readFile("document.txt");
            readFile(null);
        } catch (IOException e) {
            System.out.println("IO Exception caught: " + e.getMessage());
```

```
        }
    }
}
```

Output:

```
Reading file: document.txt
IO Exception caught: Filename cannot be null
```

Custom Exceptions

- You can create your own exception classes by extending Exception or RuntimeException.

Example: Custom Checked Exception

```
// Custom Exception Class
class InsufficientBalanceException extends Exception {
    private double amount;

    public InsufficientBalanceException(double amount) {
        super("Insufficient balance. Attempted to withdraw: $" + amount);
        this.amount = amount;
    }

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

```
// Bank Account Class
class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    public void withdraw(double amount) throws InsufficientBalanceException
{
        if (amount > balance) {
            throw new InsufficientBalanceException(amount);
        }
        balance -= amount;
        System.out.println("Withdrawal successful. Remaining balance: $" +
balance);
    }

    public double getBalance() {
        return balance;
    }
}

// Main Class
public class CustomExceptionExample {
    public static void main(String[] args) {
```

```
        BankAccount account = new BankAccount(1000.0);


        try {
            account.withdraw(500.0);  // Successful
            account.withdraw(600.0);  // Will throw exception
        } catch (InsufficientBalanceException e) {
            System.out.println("Transaction failed: " + e.getMessage());
            System.out.println("Available balance: $" + account.getBalance());
        }
    }
}
```

Output:

```
Withdrawal successful. Remaining balance: $500.0
Transaction failed: Insufficient balance. Attempted to withdraw: $600.0
Available balance: $500.0
```

Example: Custom Unchecked Exception

```
// Custom Runtime Exception
class InvalidEmailException extends RuntimeException {
    public InvalidEmailException(String email) {
        super("Invalid email format: " + email);
    }
}
// Email Validator Class
class EmailValidator {
```

```
    public static void validateEmail(String email) {
        if (email == null || !email.contains("@") || !email.contains(".")) {
            throw new InvalidEmailException(email);
        }
        System.out.println("Valid email: " + email);
    }
}

public class CustomRuntimeExceptionExample {
    public static void main(String[] args) {
        String[] emails = {"user@example.com", "invalid-email",
"another@test.org"};

        for (String email : emails) {
            try {
                EmailValidator.validateEmail(email);
            } catch (InvalidEmailException e) {
                System.out.println("Error: " + e.getMessage());
            }
        }
    }
}
```

Output:

```
Valid email: user@example.com
Error: Invalid email format: invalid-email
```

```
Valid email: another@test.org
```

Summary

Exception handling in Java is a powerful mechanism that:

- **Separates error handling from normal program logic**
- **Provides a structured way to handle runtime errors**
- **Maintains program stability and user experience**
- **Enables graceful error recovery**

Key Points to Remember:

1. Use try-catch blocks to handle exceptions
2. Handle specific exceptions rather than generic ones
3. Always clean up resources in finally blocks or use try-with-resources
4. Create custom exceptions for specific business logic errors
5. Provide meaningful error messages
6. Don't ignore exceptions - log or handle them appropriately
7. Use checked exceptions for recoverable conditions
8. Use unchecked exceptions for programming errors

Exception handling is essential for writing robust, maintainable Java applications that can gracefully handle unexpected situations and provide meaningful feedback to users and developers.