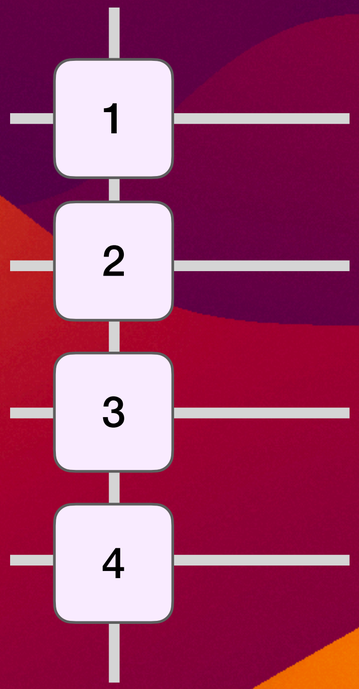
Exception Handling in Java

Fundamentals, Exception Types, Handling Techniques

What is Exception Handling?



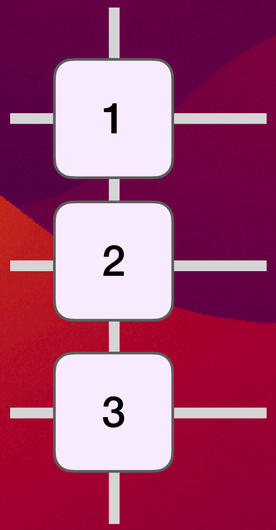
Mechanism to handle runtime errors.

Maintains the flow of execution.

Helps in identifying and managing error-prone code.

Improves code robustness and reliability.

Importance of Exception Handling



Prevents abrupt program termination.

Allows graceful recovery from errors.

Essential for real-world applications (file handling, networking, etc.).

Categories of Exceptions

1. **Checked Exceptions**: Known at compile time (e.g., IOException).

2. **Unchecked Exceptions**: Occur during runtime (e.g., ArithmeticException).

3. **Errors**: Critical issues beyond the control of the program (e.g., OutOfMemoryError).

Difference Between Checked and Unchecked Exceptions

1. **Checked**: Must be handled using `try-catch` or declared with `throws`.

2. **Unchecked**: Can be left unhandled but may cause runtime failures.

3. **Error Class**: Should not be caught or handled directly.

Exception Hierarchy in Java

Exception Hierarchy

1. Root class: `Throwable`.
2. Subclasses: `Exception` and `Error`.
3. **Exception**: Further divided into checked and unchecked exceptions.
4. **Error**: Represents system-level issues.

Handling Exceptions

Exception Handling Keywords

1. **try**: Block where exceptions may occur.
2. **catch**: Handles the exception.
3. **finally**: Executes after try-catch, regardless of outcome.
4. **throw**: Manually throw an exception.
5. **throws**: Declare an exception for a method.

Caught vs Uncaught Exceptions

Caught and Uncaught Exceptions

1. **Caught Exception**: Handled using `try-catch`.

2. **Uncaught Exception**: Propagates up the call stack until handled or causes the program to terminate.

Using try-catch-finally

try-catch-finally Example

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Handle the exception
} finally {
    // Cleanup code
}
```

The throw Keyword

Using the throw Keyword

1. Used to manually throw an exception.
2. Example: throw new ArithmeticException("Division by zero");

The throws Clause

1. Used in method declaration to specify exceptions that the method might throw.
2.
```
public void readFile() throws IOException {
    // Code that may cause an IOException
}
```

Common Built-in Exceptions

1. **NullPointerException**: Occurs when trying to access an object with a null reference.
2. **ArrayIndexOutOfBoundsException**: Accessing an array with an invalid index.
3. **ClassCastException**: Invalid casting of objects.
4. Define your own exception by extending the Exception class.
5. Usage: throw new CustomException("Custom error message”);

```
class CustomException extends Exception {
    CustomException(String message) {
        super(message);
    }
}
```

Common Built-in Exceptions

```
class AgeException extends Exception {
    AgeException(String message) {
        super(message);
    }
}

public class TestAgeException {
    static void validate(int age) throws AgeException {
        if (age < 18) {
            throw new AgeException("Not eligible for
voting");
        }
    }
```

```
    public static void main(String[] args) {
        try {
            validate(16);
        } catch (AgeException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

Best Practices

1. Always catch specific exceptions, not the generic `Exception`.
2. Avoid using exceptions for control flow.
3. Clean up resources in `finally`.
4. Properly document exceptions thrown by methods

Summary of Exception Handling

1. Java provides a structured mechanism for handling exceptions.
2. Use `try-catch-finally` for handling exceptions.
3. Distinguish between checked and unchecked exceptions.
4. Custom exceptions provide more meaningful error handling.

Thank You