Error Handling

Topcis	**Try-catch-finally block | Throwing custom exceptions | Printing stack trace**

Module: Error Handling & Asynchronous Programming

Class: Error Handling

Topics: Try-Catch-Finally | Custom Exceptions | Stack Traces

1. Lesson Objectives

By the end of this session, students will be able to:

- Implement robust error handling using **try-catch-finally** blocks.
- Differentiate between standard Errors and Exceptions.
- Create and throw **Custom Exceptions** for specific application logic.
- Utilize **Stack Traces** to debug and locate the origin of errors effectively.

2. Core Concepts: Handling Errors

In Dart (and Flutter), proper error handling prevents your application from crashing unexpectedly and allows you to provide user-friendly feedback when things go wrong.

The Try-Catch-Finally Block

- **Try:** Wraps the code that *might* throw an exception (e.g., fetching data, parsing JSON).
- **Catch:** Catches the exception if one occurs, preventing a crash. You can catch specific types of exceptions or a general one.

- **Finally:** A block that *always* runs, regardless of whether an error occurred or not. It is typically used for cleanup (e.g., closing files, resetting loading states).

The Stack Trace

A **Stack Trace** is a snapshot of the call stack at the moment the error occurred. It tells you exactly *where* (which file and line number) the error happened and the sequence of function calls that led there.

Lecture Code Example

```
void main() {
  print("App Started");

  try {
    // 1. Risky code
    var result = 10 ~/ 0; // Integer division by zero throws an Exception
    print("Result: $result");

  } on IntegerDivisionByZeroException {
    // 2. Catching a specific exception
    print("Cannot divide by zero!");

  } catch (e, s) {
    // 3. Catching any other error
    // 'e' is the Exception object
    // 's' is the StackTrace object
    print("Unknown error: $e");
    print("Stack Trace:\n$s");

  } finally {
    // 4. Cleanup code
    print("Cleanup operations (closing streams/files)...");
  }

  print("App Continued..."); // App survives the error
```

```
}
```

3. Throwing Custom Exceptions

Sometimes standard exceptions (like `FormatException` or `TimeoutException`) aren't specific enough for your business logic. You can define your own.

How to Create a Custom Exception

Create a class that implements the built-in `Exception` interface.

Scenario

Imagine a user trying to withdraw money from a bank account. We need a specific error if they don't have enough funds.

Code Example

```
// 1. Define the Custom Exception
class InsufficientFundsException implements Exception {
  final String message;
  final double attemptedAmount;

  InsufficientFundsException(this.message, this.attemptedAmount);

  @override
  String toString() => "InsufficientFundsException: $message (Attempted: \$$at
temptedAmount)";
}

class BankAccount {
  double balance = 100.0;

  void withdraw(double amount) {
    if (amount > balance) {
      // 2. Throw the custom exception
```

```
      throw InsufficientFundsException("Not enough balance", amount);
    }
    balance -= amount;
    print("Withdrew \$$amount. Remaining: \$$balance");
  }
}

void main() {
  var account = BankAccount();

  try {
    account.withdraw(500.0); // This will fail
  } catch (e) {
    // 3. Handle the specific custom logic
    if (e is InsufficientFundsException) {
      print("Transaction failed: ${e.message}");
      // specific logic for this error, e.g., show a 'Top Up' dialog
    } else {
      print("General Error: $e");
    }
  }
}
```

4. Key Takeaways

1. **Don't Let Apps Crash:** Always wrap risky code (network calls, file I/O) in `try-catch` blocks to keep the app running.
2. **Be Specific:** Catch specific exceptions (using `on ExceptionName`) before catching general ones. This allows you to handle different errors differently (e.g., "No Internet" vs. "Invalid Password").
3. **The `finally` Clause:** Crucial for UI logic. For example, if you show a loading spinner in `try`, you *must* hide it in `finally` so it doesn't spin forever if an error occurs.

4. **Debug with Stack Traces:** When catching a generic error, always print the stack trace (`catch (e, s)`) during development to find the bug's origin quickly.