MLA// EXTENDED ASSESSMENT TASK

WEEK // SESSION	WEEK THREE // SESSION SIX
NAME OF SESSION	DEBUGGING AND ERROR HANDLING

ASSESSMENT FOCUS	Debugging in a Real-Life Context

Scenario:

You've recently implemented a new feature into your Fitness Tracker App; however, several users have reported that the app crashes when they try to use this new feature. You suspect there might be an error in your code.

Task:

1. **Debugging Strategies:** Describe the debugging techniques you would employ to identify the root cause of the crashes.

2. **Error Handling:** Explain how you would implement error handling mechanisms to handle these situations, prevent crashes, and provide informative messages to the user.

3. **Defensive Programming:** Explain how you could have applied the principles of defensive programming to potentially prevent this error in the first place. Provide specific examples of defensive programming techniques you would use when writing code for your fitness tracker app.

Each section should be approximately 200 words.

Debugging Strategies

To identify the root cause of crashes in the new fitness tracker feature, I would employ a systematic debugging approach starting with data collection. First, I would examine crash logs and stack traces from user reports to identify the exact point of failure and error messages. I would then attempt to reproduce the crash in a controlled development environment by replicating the exact user actions and data inputs that trigger the failure.

Next, I would use browser developer tools or debugger breakpoints to step through the code execution line-by-line, monitoring variable states and data flow at each stage. This helps identify where values become undefined or unexpected. I would also implement strategic console.log statements at critical points - before and after API calls, during data transformations, and at conditional branches - to trace the execution path.

Additionally, I would check for common issues like null/undefined values, asynchronous timing problems, improper error handling in API responses, and data type mismatches. Testing with edge cases (empty datasets, extreme values, special characters) often reveals issues missed during normal testing. I would also review recent code commits using Git to identify what changed between the working and broken versions. Finally, gathering device-specific information from crash reports helps identify if the issue is platform-specific or affects all users universally.

Error Handling

To prevent crashes and provide better user experience, I would implement comprehensive error handling throughout the feature. First, I would wrap all critical operations in try-catch blocks to gracefully catch exceptions before they crash the application. For example, when processing fitness data submissions, the catch block would log the error details while displaying a user-friendly message like "Unable to save workout. Please try again."

For API calls to backend services, I would implement proper response validation - checking if responses exist and contain expected data structures before attempting to use them. If the API fails, instead of crashing, the app would show: "We're having trouble connecting. Please check your internet connection."

I would also add input validation on the frontend before sending data to prevent invalid data from causing backend errors. For instance, validating that workout duration is a positive number and date fields contain valid dates.

Additionally, I would implement a global error boundary in React components to catch rendering errors and display a fallback UI instead of a blank screen. All error messages would be logged to a monitoring service like Sentry for debugging while showing users actionable, non-technical messages. Critical errors would trigger automatic bug reports with context about what the user was attempting, helping developers quickly identify patterns and fix issues.

Defensive Programming (200 words)

Applying defensive programming principles from the start could have prevented many potential crashes. First, I would implement strict input validation at every entry point - checking that all user inputs match expected formats, ranges, and types before processing. For the fitness tracker, this means validating that exercise duration is a positive number, dates fall within reasonable ranges, and required fields aren't empty or null.

I would use explicit null/undefined checks before accessing object properties. For example: `if (user && user.profile && user.profile.weight) { ... }` or optional chaining: `user?.profile?.weight` to safely navigate nested objects without crashing if intermediate properties don't exist.

Type safety would be enforced using TypeScript instead of plain JavaScript, catching type-related errors at compile time rather than runtime. Defining interfaces for data structures ensures consistency throughout the application.

I would also implement default values and fallbacks everywhere - if expected data is missing, the app uses sensible defaults rather than breaking. For instance, if a user's exercise history fails to load, display an empty state UI instead of crashing.

Additionally, I would write comprehensive unit tests covering normal cases, edge cases, and error conditions before deploying features. Test-driven development ensures code handles unexpected scenarios gracefully. Finally, implementing proper logging at key points helps track data flow and quickly identify issues when they occur in production.