Good Fortune Teller Implementation

This implementation follows clean coding practices, avoids unnecessary features, and separates responsibilities properly.

1. **Clean Code**:
   - **Descriptive variable names**: Fortunes are stored in a list called fortunes, making it a good name.
   - **Clear structure**: The code is organized into functions with distinct purposes, improving readability and maintainability.
2. **Single Responsibility Principle**:
   - **Function separation**: Each function serves a single, clear purpose. For example, get_fortune() only handles selecting a fortune, and is_legal_age() checks the user's age.
3. **YAGNI (You Aren't Gonna Need It)**:
   - **Avoids unnecessary features**: The code avoides extra input prompts or unrelate functions.

Bad Fortune Teller Implementation

This bad version of the code violates key coding practices, making it a poor example of good software design.

1. **Clean Code Violation**:
   - **Poor variable names**: Variables like rick_and_morty_season_6_5 and johhny_boy are confusing and unrelated to the functionality, making the code hard to understand.
   - **Irrelevant comments**: Comments are either unnecessary or do not add meaningful explanations to the code, cluttering the implementation.
2. **Single Responsibility Principle Violation**:
   - **Too many tasks in one function**: The main function bad_fortune_teller() handles multiple responsibilities, including input gathering, age validation, fortune display, and handling program restarts, all in one place.
3. **YAGNI Violation**:
   - **Unnecessary features and code**: The code uses functions that are not used or needed (extra_feature_unused and days_since_fortune), adding complexity without value.