

Developer Guide

Format inspired by [addressbook-level3](#)

Table of Contents

- [Acknowledgements](#)
- [Design](#)
 - [Architecture](#)
 - [UI Component](#)
 - [Responsibilities](#)
 - [Key classes](#)
 - [Input processing flow](#)
 - [Notes](#)
 - [Logic and Commands component](#)
 - [Controller component](#)
 - [Model component](#)
 - [Storage component](#)
 - [Common classes](#)
 - [Program Flow](#)
 - [Main Execution Loop](#)
 - [Storage Operations](#)
- [Product scope](#)
 - [Target user profile](#)
 - [Key features \(current / planned\)](#)
 - [Brainstorming and references](#)
 - [Non-goals \(out of scope for current release\)](#)
- [Implementation](#)
 - [CompleteExercise feature](#)
 - [CompleteSession feature](#)
 - [DeleteAthlete feature](#)
 - [DeleteExercise feature](#)
 - [DeleteSession feature](#)
 - [Exit feature](#)
 - [FlagAthlete feature](#)
 - [Leaderboard feature](#)
 - [ListAthlete feature](#)
- [Testing](#)
- [Appendix E: Instructions for Manual Testing](#)
 - [Launch and shutdown](#)
- [Instructions for Manual Testing \(storage\)](#)
 - [Product scope](#)
 - [Target User profile](#)
 - [Value Proposition](#)
 - [User Stories](#)

- [Use cases](#)
 - [Use Case: Add a New Athlete](#)
 - [Use Case: Delete an Athlete](#)
 - [Use Case: Add a New Session](#)
 - [Use Case: Delete a Session](#)
 - [Add an Exercise](#)
 - [Use Case: Delete an Exercise](#)
 - [Use Case: Complete an Exercise](#)
 - [Use Case: Complete a Session](#)
 - [Use Case: Undo an Exercise](#)
 - [Use Case: Complete a Session](#)
 - [Use Case: Flag an Athlete](#)
 - [Use Case: List All Athletes](#)
 - [Use Case: View Athlete Details](#)
 - [Use Case: View All Sessions for an Athlete](#)
 - [Use Case: View All Exercises in a Session](#)
 - [Use Case: Update Session Notes](#)
 - [Use Case: View Help \(List All Commands\)](#)
 - [Use Case: View Leaderboard](#)
 - [Use Case: Persist and Restore Coach Data](#)
- [Non-Functional Requirements](#)
- [Glossary](#)

Acknowledgements

We would like to express our gratitude to:

- Professor **Akshay Narayan** for his guidance and supervision of this project
- Teaching Assistant **Hing Yen Xing** for their valuable feedback and support

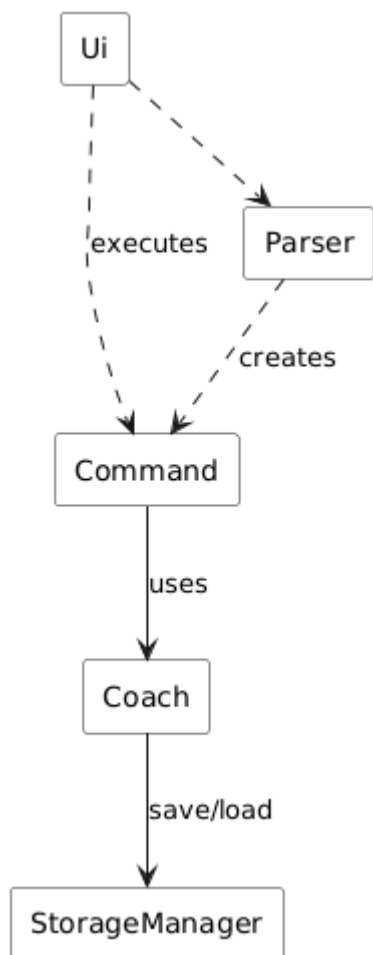
And our team members who contributed to this project:

- **Halil Cokeren**
- **Toh Ee Sen Izen**
- **Yeung Ho / Gordon**
- **Philip Hansson**
- **Ma Zhiheng**

Design

Architecture

The Architecture section explains the high-level design of FitnessONE and how its components collaborate to process commands, manipulate domain state, and persist data.



PlantUML source (kept under version control): [diagrams/architecture.puml](#).

The architecture is loosely inspired by MVC:

- View → **Ui**
- Controller → Commands + **Coach**
- Model → **Athlete**, **Session**, **Exercise**

The diagram above summarizes the main runtime components and their dependencies.

Responsibilities

- Provide a command-line interface (CLI) for user interaction
- Read raw user input, display formatted output and error messages
- Delegate input parsing to **seedu.fitnessone.ui.Parser**
- Forward parsed commands to the Logic component and display command results
- UI does not contain business logic; it only formats input/output and handles presentation concerns

Key Classes

- **seedu.fitnessone.ui.Parser** — converts raw input strings to command objects and validates basic syntax
- **Ui** / **TextUi** — reads input from stdin, prints to stdout, and formats messages
- Any **Message** or **UiStrings** class — centralises user-facing strings and error messages
- UI: Reads input and prints formatted output (package **seedu.fitnessone.ui**).

- Logic: Parses commands and executes them via Command classes (package `seedu.fitnessone.command`).
- Controller: Coordinates model operations and encapsulates domain rules (class `seedu.fitnessone.controller.Coach`).
- Model: Domain entities and in-memory state (package `seedu.fitnessone.model` — `Athlete`, `Session`, `Exercise`).
- Storage: Loads and saves state to the text file store (package `seedu.fitnessone.storage` — `StorageManager`).
- Common/Exceptions: Reusable types and exception hierarchy (package `seedu.fitnessone.exception`).

Input Processing Flow

1. UI reads input and passes it to `Parser.parse(...)`
2. Parser creates a Command object which is passed to the Logic component
3. For invalid input or domain errors (e.g. `InvalidAthleteException`), UI catches exceptions and prints user-friendly messages:

```
Error: Athlete not found - 0001
Caused by: seedu.fitnessone.exception.InvalidAthleteException: Invalid
Athlete ID: 0001
```

How components interact

1. `FitnessONE` (entry point) starts the app, constructs `Ui`, `Parser`, `Coach`, and `StorageManager`.
2. On startup, `StorageManager.load()` reconstructs the `Coach` state from `data/athletes_export.txt`.
3. The main loop reads a line of user input from `Ui`, then `Parser.parse(...)` returns a `Command` instance.
4. `Command.execute(coach, ui)` invokes controller/model methods to perform the requested operation.
5. On successful commands that mutate state, `StorageManager.save(coach)` persists the new state.
6. Exceptions from the controller/model are translated to user-friendly messages and printed by `Ui`.

Sequence at a glance

- Command processing loop: see Program Flow → Main Execution Loop.
- Persistence: see Program Flow → Storage Operations (startup load and per-command save).

Design choices

- Keep UI thin; formatting and I/O only. Business logic lives in Commands and the Coach controller.
- Centralize domain rules (limits, ID assignment, lookups) in `Coach` to avoid duplication across commands.
- A simple line-oriented storage format ensures deterministic round-trips (`ATHLETE|...`, `SESSION|...`, `EXERCISE|...`).

UI Component

The UI is CLI-based and is responsible for interacting with stdin/stdout. It doesn't contain business logic.

Responsibilities

- Read raw user input lines
- Print success/error messages with consistent dividers
- Delegate parsing to `Parser` and execution to `Command`

Key classes

- `seedu.fitnessone.ui.Ui` — console I/O helpers and message formatting
- `seedu.fitnessone.ui.Parser` — converts raw strings to concrete `Command` objects; validates basic syntax and IDs

Input processing flow

1. `Ui` reads a line.
2. `Parser.parse(...)` inspects the leading token (e.g., `/newathlete`) and constructs a concrete `Command`.
3. Errors in syntax or missing parameters raise `InvalidCommandException`, which the app prints via `Ui.printWithDivider`.

Notes

- Command-specific help is available by triggering a command with missing/invalid parameters.

Logic and Commands component

Responsibilities

- Interpret parsed input into domain actions
- Validate parameters and preconditions (existence of IDs, argument counts)
- Execute operations by calling the controller/model and return user-facing messages

Key types

- `seedu.fitnessone.command.Command` — base class with `execute(Coach coach, Ui ui)`
- Concrete commands e.g., `NewAthleteCommand`, `ViewAthleteCommand`, `ViewSessionsCommand`, `ViewExerciseCommand`, `CompleteSessionCommand`, `CompleteExerciseCommand`, `DeleteAthleteCommand`

Execution contract

- Input: references to the live `Coach` and `Ui`, plus any arguments captured during parsing
- Output: side-effects on the model; messages printed via `Ui`
- Error modes: throws `InvalidCommandException` (syntax/args) or propagates domain exceptions from controller/model

Controller component

Responsibilities

- Provide a single façade (`seedu.fitnessone.controller.Coach`) over domain data
- Enforce domain rules and limits (e.g., capacity limits; valid lookups)
- Allocate and manage identifiers; provide indexed/filtered accessors

Notes

- Commands call `Coach` methods for retrieving/updating `Athlete`, `Session`, and `Exercise` objects.
- Domain exceptions thrown here bubble up to the UI with user-friendly messages.

Model component

Responsibilities

- Represent domain entities and in-memory state for athletes, sessions, and exercises
- Provide simple getters/setters and minimal invariants (e.g., completed flags)

Key classes (package `seedu.fitnessone.model`)

- `Athlete` — name and collections of `Session`
- `Session` — metadata and a list of `Exercise`; has a `completed` flag
- `Exercise` — description, sets/ reps and a `completed` flag

Identifiers

- IDs are centrally managed by `Coach`. Commands never construct IDs directly; they obtain and use IDs via controller/model APIs.

Storage component

Responsibilities

- Persist and load the entire application state to/from the filesystem
- Provide deterministic round-trip serialization for `Coach`, `Athlete`, `Session`, and `Exercise`

Key class

- `seedu.fitnessone.storage.StorageManager` — handles `load()` on startup and `save(coach)` after successful commands

Format and location

- Text-based, line-oriented format stored under `data/athletes_export.txt`
- Representative records: `ATHLETE | ... , SESSION | ... , EXERCISE | ...`

See also

- Program Flow → Storage Operations for startup and runtime sequences
- Implementation → `StorageManager` details for edge cases and error handling

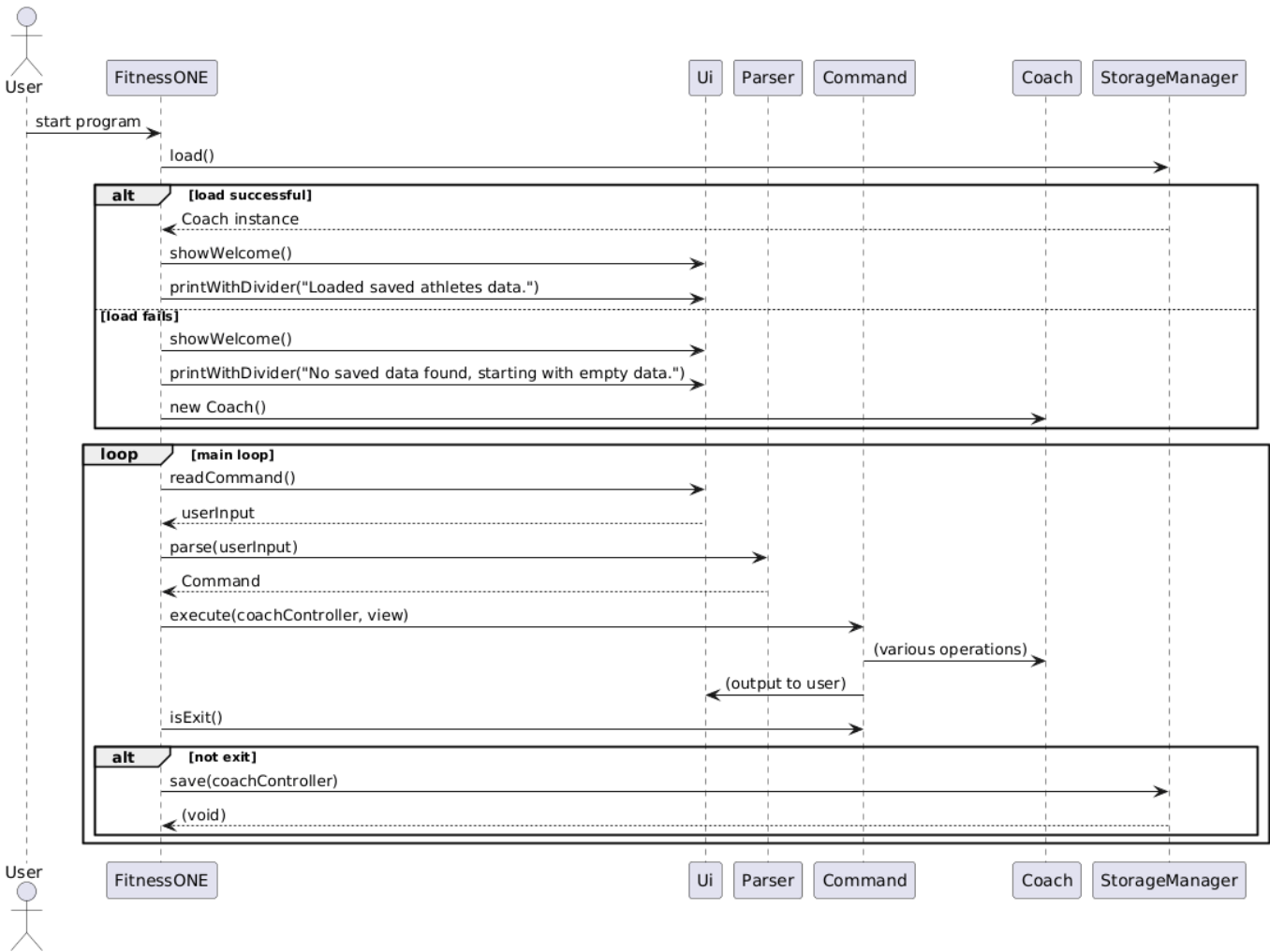
Common classes

- Exceptions: `seedu.fitnessone.exception.*` define user-visible failures (invalid IDs, limits reached, I/O errors)
- Messages: user-facing strings are centralized in the UI/command layer for consistent formatting

Class diagram (PlantUML source): [diagrams/components_class.puml](#)

Program Flow

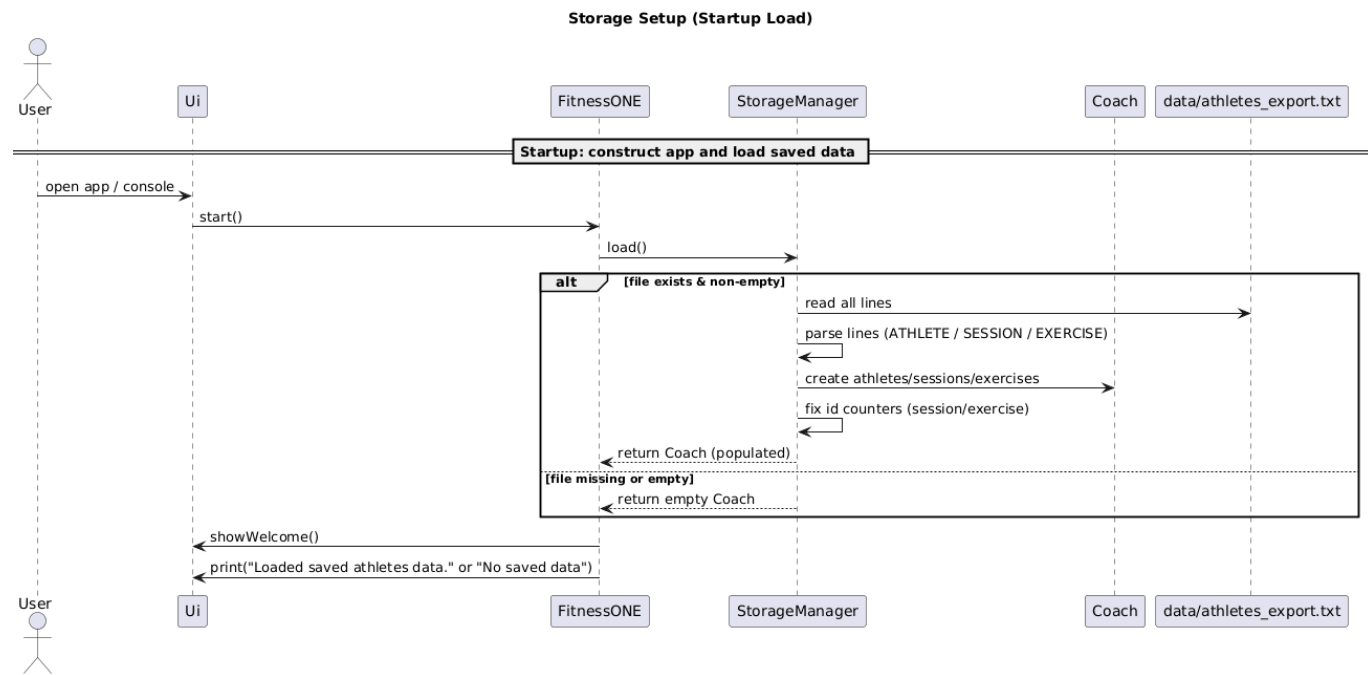
Main Execution Loop



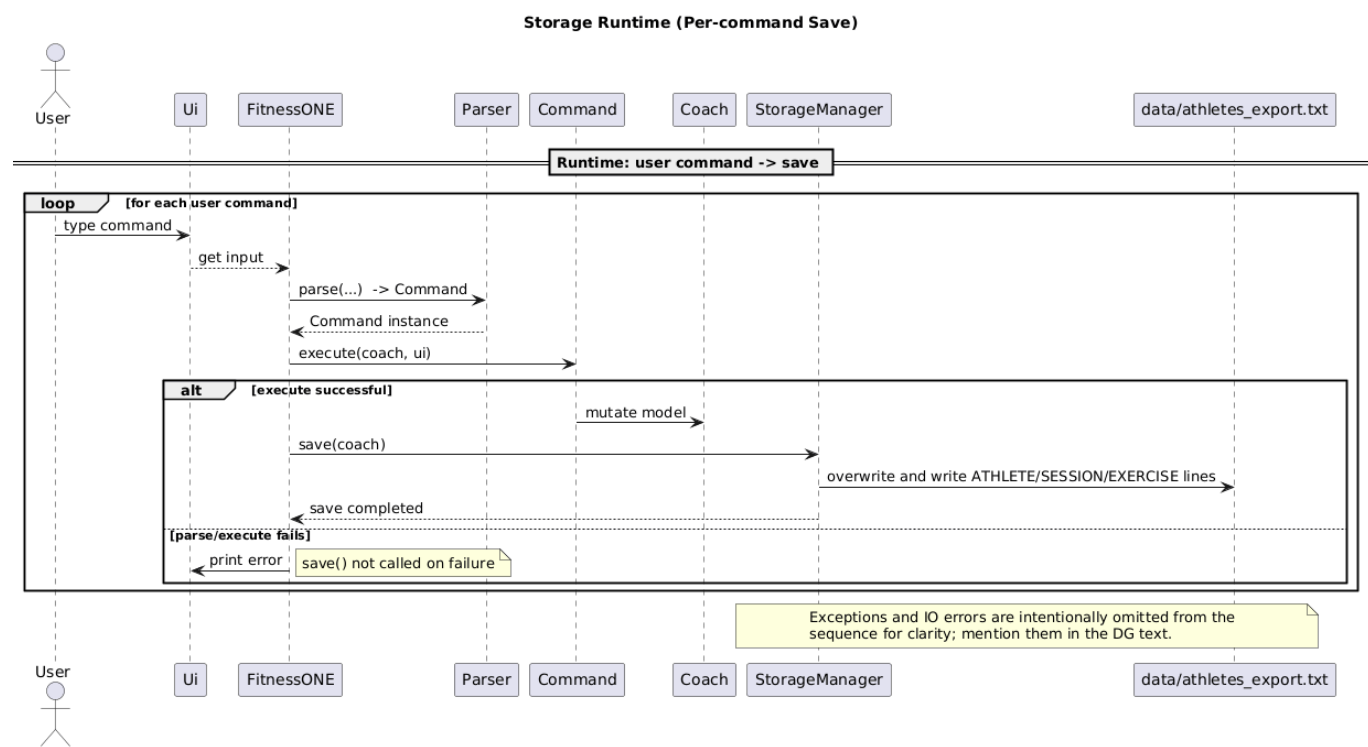
The sequence diagram above shows how FitnessONE processes commands in its main loop, from user input through execution and persistence.

Storage Operations

Startup (load on app start):



Runtime (save after each successful command):



These diagrams illustrate the data persistence flow: the first covers startup loading behavior, and the second shows per-command saving operations.

Product scope

Product name: FitnessONE

Target user profile

FitnessONE is designed for professional fitness coaches who need a compact, scriptable CLI tool to manage and monitor multiple students' training routines and nutrition. Typical users:

- Individual coaches running small teams or classes
- Strength & conditioning coaches tracking sessions and macros
- Coaches who prefer lightweight CLI tools and version-controlled data

Formatting and examples

- Keep prompts and responses simple and consistent. Example interaction:
 - Input: `/viewAthlete 0001`
 - Output: `Showing athlete 0001: <name> – <summary>`
 - Error: `Error: Athlete not found – 0001`

FitnessONE provides coaches an efficient, data-driven platform for exercise logging, macronutrient tracking, and automatic recommendations for diets and exercises. It is simple to use and integrates persistent storage so coaches can generate progress reports and plan long-term training efficiently.

Key features (current / planned)

- Track athletes (create, view, update athlete records)
- Track training sessions per athlete (notes, completed flag)
- Track exercises within sessions (description, sets, reps, completed)
- Track macronutrients (protein/carbs/fats) per athlete and per day (planned)
- Recommend diet plans and exercise suggestions based on logged data (planned)
- Persistent storage to `data/athletes_export.txt` (load on startup, save after successful commands)
- Import/export and round-trip loading for reproducible testing

Brainstorming and references

- Week 5 brainstorming notes (planning and options considered):
<https://docs.google.com/spreadsheets/d/1CClecnanB5N0eg2bsRAJShKHbCECX9t7Hzhew0F9al/edit?usp=sharing>
- Input model options considered:
 - Predetermined plans (pre-built programs coaches can assign)
 - Fully user-input exercises (coach enters exercise name, sets, reps, weight)

Non-goals (out of scope for current release)

- Real-time sync across devices or multi-user concurrent editing
- Complex GUI client (this is a CLI-first tool)

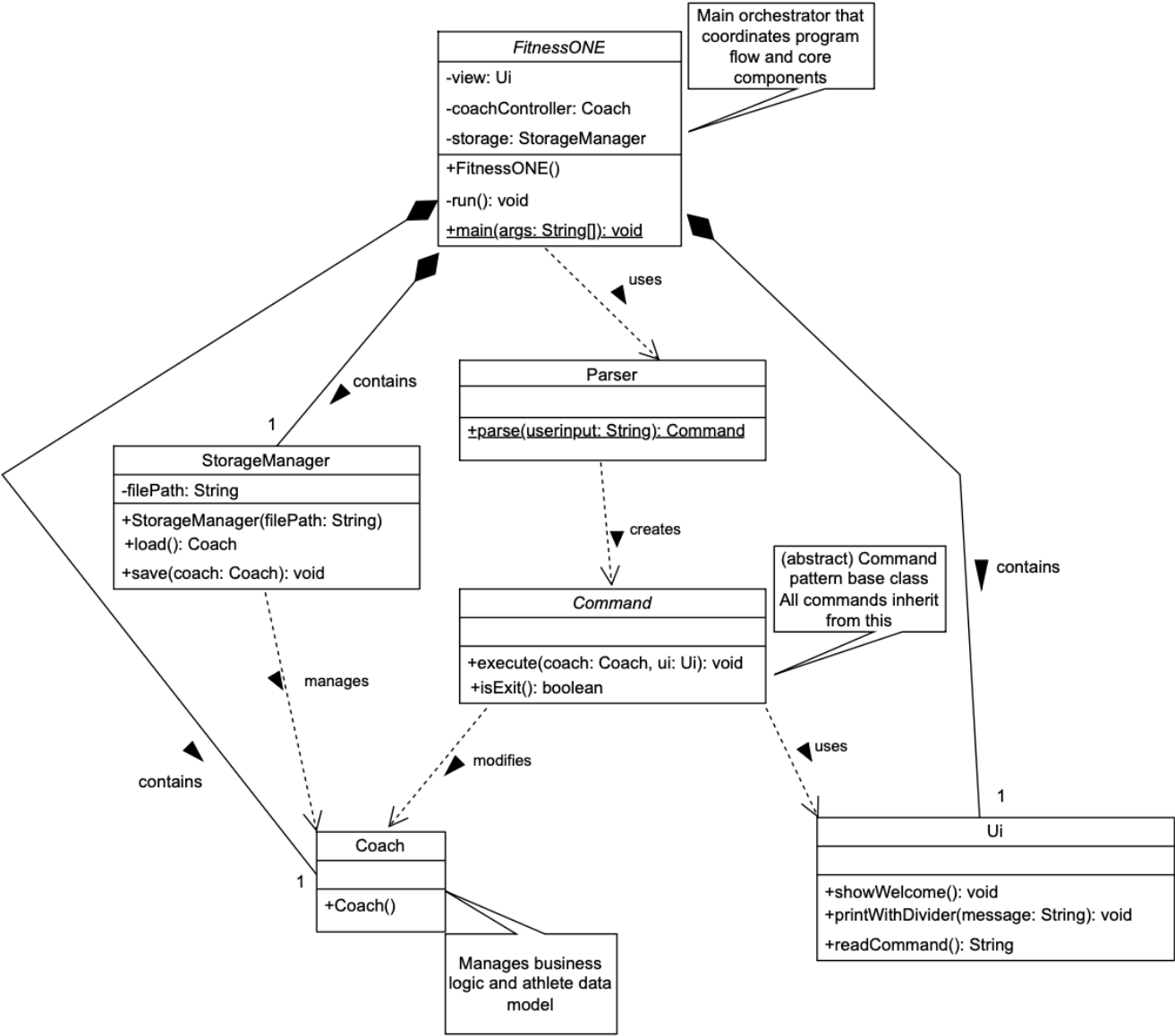
Design notes

- Keep UI code minimal so changes to presentation do not affect logic.
- Centralise all user strings to make updating messages and translations easier.

Version	As a ...	I want to ...	So that I can ...
<hr/>			

Version	As a ...	I want to ...	So that I can ...
v1.0	coach	create and manage athlete records; add training sessions and exercises; log daily macronutrients	track each student's training and nutrition data and preserve session history
v2.0	coach	receive diet and exercise recommendations; export/import athlete data	adapt plans automatically based on tracked data and share/archive team progress

Implementation

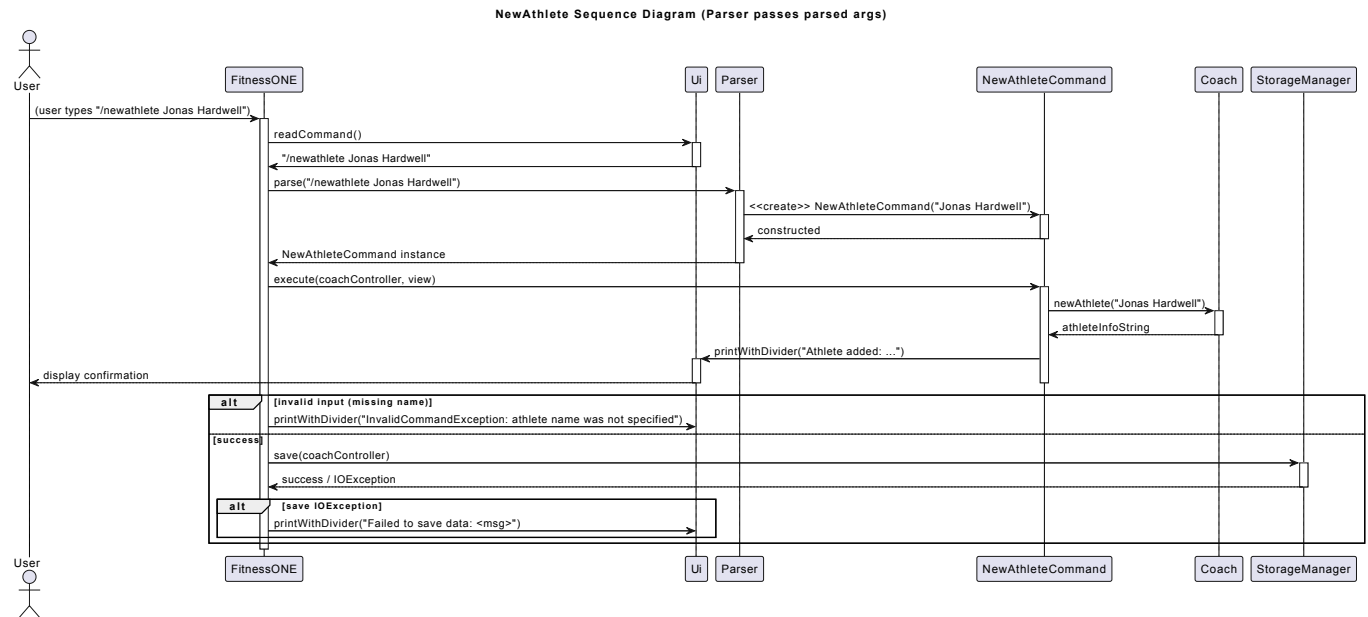


This section describes some noteworthy details on how certain features are implemented. In general, the caller is the run method in FitnessONE; it reads user input, passes it to Parser.parse() to obtain a Command, invokes Command.execute(), then persists state.

NewAthlete feature

Purpose: Show how the application handles a user request to create a new athlete, which is implemented by NewAthleteCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls view.readCommand() to get the raw input string from the user (Ui.readCommand).
2. FitnessONE passes the raw string to Parser.parse(...).
3. Parser identifies the command keyword /newathlete, parses the arguments (e.g., "Jonas Hardwell"), and constructs a NewAthleteCommand instance via new NewAthleteCommand("Jonas Hardwell").
4. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by FitnessONE and dynamically dispatched to NewAthleteCommand.execute.
5. Inside execute, NewAthleteCommand calls coachController.newAthlete("Jonas Hardwell") to create a new athlete and add it to the current coach's athlete list.
6. NewAthleteCommand then prints a confirmation message through view.printWithDivider, showing that the athlete has been successfully added.
7. Control returns to FitnessONE.run; the method then persists the updated coach data by calling storage.save(coachController) and reports any I/O errors to the Ui if necessary.

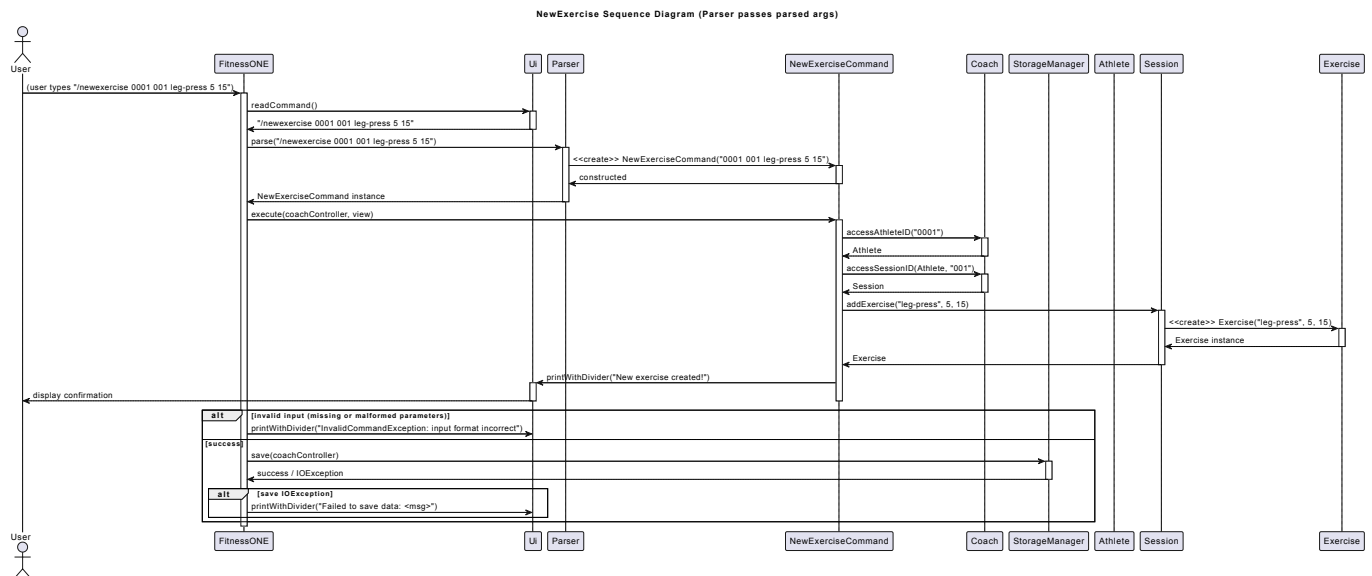
Error handling:

1. If the command arguments are missing (e.g., athlete name not provided), Parser throws an InvalidCommandException, which is caught and displayed by Ui.printWithDivider.
2. If the creation fails due to an internal logic error, NewAthleteCommand.execute throws an exception that is similarly caught and displayed.
3. FitnessONE.run does not call save when command execution fails.

NewExercise feature

Purpose: show how the application handles a user request to add a new exercise to an existing session, which is implemented by NewExerciseCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
2. FitnessONE passes the raw string to Parser.parse(...), which parses the arguments (Athlete ID, Session ID, Exercise Description, sets, reps) and constructs a NewExerciseCommand instance.
3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to NewExerciseCommand.execute.
4. NewExerciseCommand.execute requests the Coach for the Athlete (coachController.accessAthleteID) and then for the Session (coachController.accessSessionID).
5. The command calls session.addExercise(description, sets, reps) to create a new Exercise object.
6. NewExerciseCommand prints confirmation via view.printWithDivider, displaying the newly added exercise details along with Athlete and Session info.
7. Control returns to FitnessONE.run; run persists state with storage.save(coachController) and reports any I/O errors to the Ui.

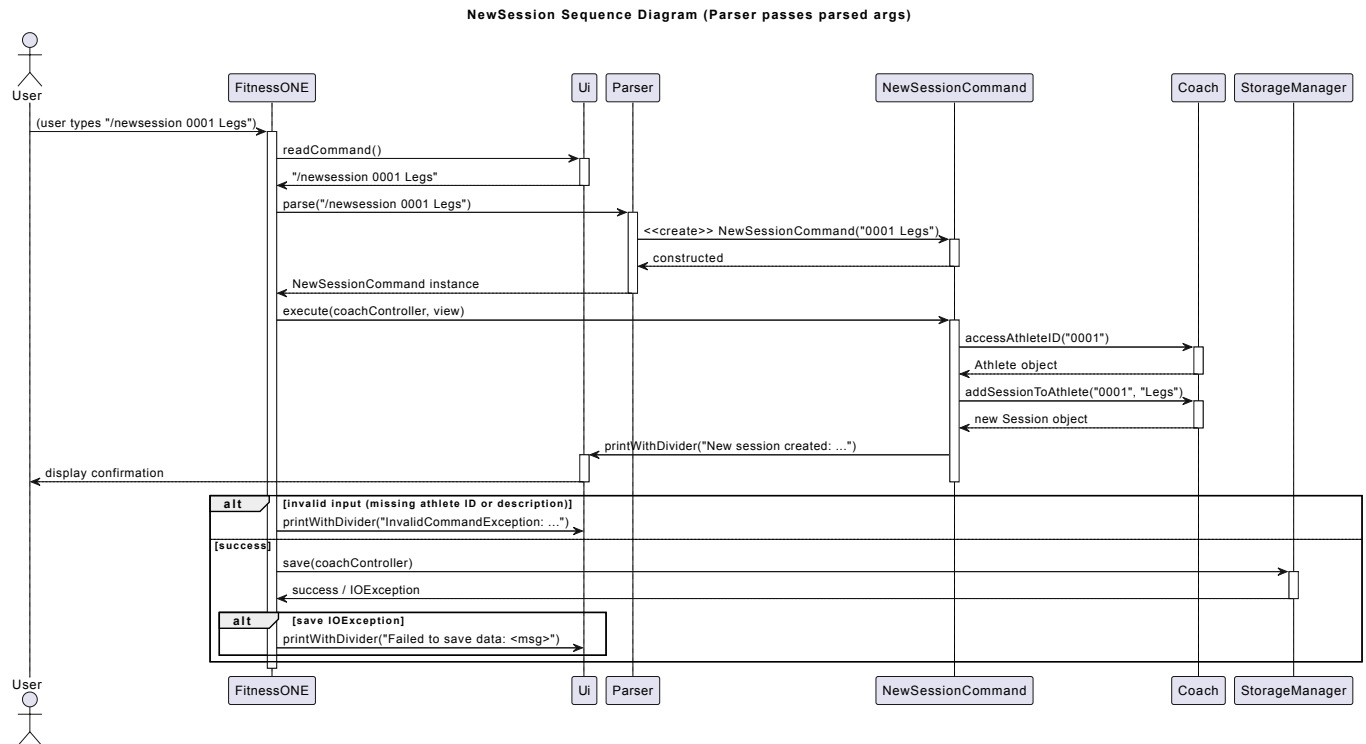
Error handling:

1. If the Athlete or Session is not found, Coach throws InvalidAthleteException or InvalidSessionException.
2. If the input parameters are invalid (e.g., missing fields or sets/reps are not integers), NewExerciseCommand throws InvalidCommandException.
3. These exceptions are caught and wrapped/displayed via Ui.printWithDivider.
4. FitnessONE.run does not call save on failure.
5. If saving fails due to an IOException, Ui.printWithDivider displays the error message.

NewSession feature

Purpose: show how the application handles a user request to add a new session for an existing athlete, which is implemented by NewSessionCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
2. FitnessONE passes the raw string to Parser.parse(...), which constructs a NewSessionCommand instance with the parsed parameters.
3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to NewSessionCommand.execute.
4. NewSessionCommand.execute calls coachController.accessAthleteID(athleteID) to get the Athlete object.
5. The command then calls coachController.addSessionToAthlete(athleteID, trainingNotes) to create a new Session object and add it to the athlete.
6. NewSessionCommand prints a confirmation message via view.printWithDivider(...).
7. Control returns to FitnessONE.run; run persists state with StorageManager.save(coachController) and reports any I/O errors to the Ui.

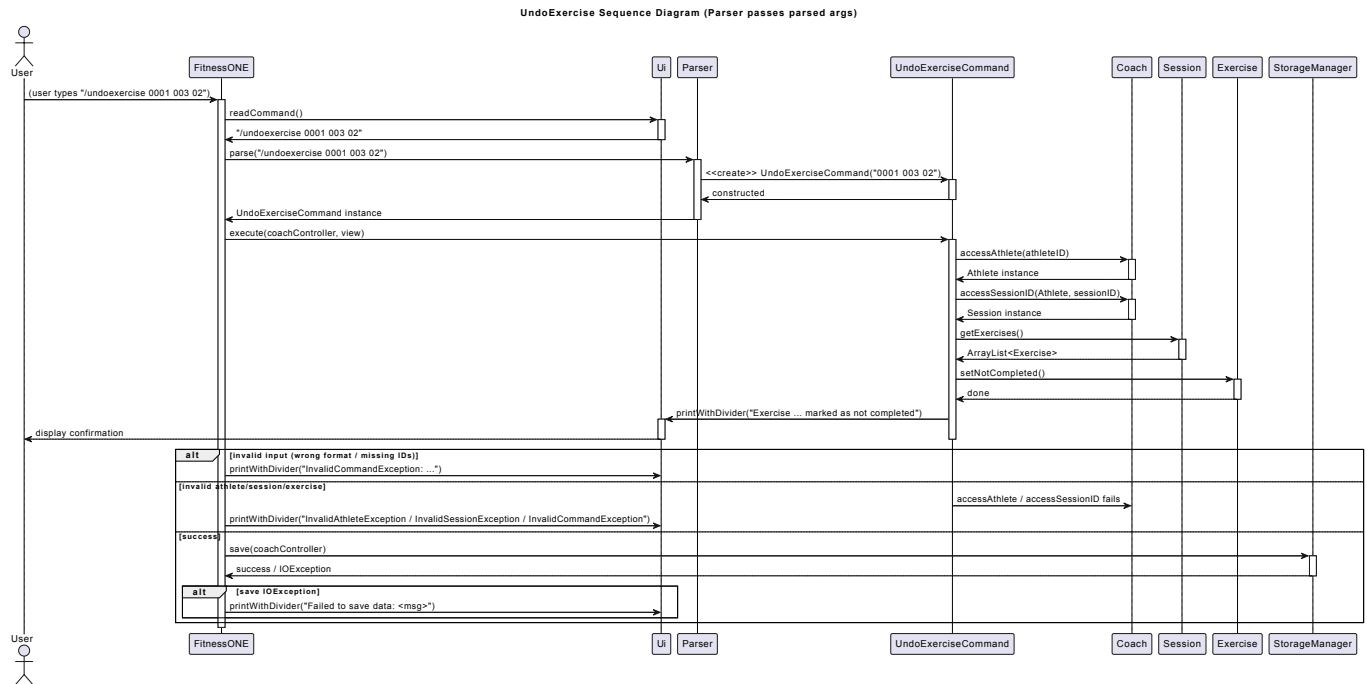
Error handling:

1. If the athlete ID is invalid or missing, Coach throws InvalidAthleteException; if the command input is malformed, NewSessionCommand throws InvalidCommandException.
2. FitnessONE wraps these into messages displayed via Ui.printWithDivider(...).
3. If saving via StorageManager fails (IOException), the error is displayed via Ui.printWithDivider(...).
4. FitnessONE.run does not call save on failure.

UndoExercise feature

Purpose: show how the application handles a user request to mark a previously completed exercise as not completed, implemented by UndoExerciseCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
2. FitnessONE passes the raw string to Parser.parse(...), which constructs an UndoExerciseCommand instance.
3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by FitnessONE and dynamically dispatched to UndoExerciseCommand.execute.
4. UndoExerciseCommand.execute: it calls Coach.accessAthlete(athleteID) to get the Athlete, then Coach.accessSessionID(Athlete, sessionID) to get the Session.
5. The command iterates session.getExercises(), matches the exerciseID, and calls exercise.setNotCompleted() when found.
6. The command prints confirmation via view.printWithDivider.
7. Control returns to FitnessONE.run; run persists state with StorageManager.save(coachController) and reports any I/O errors to the Ui.

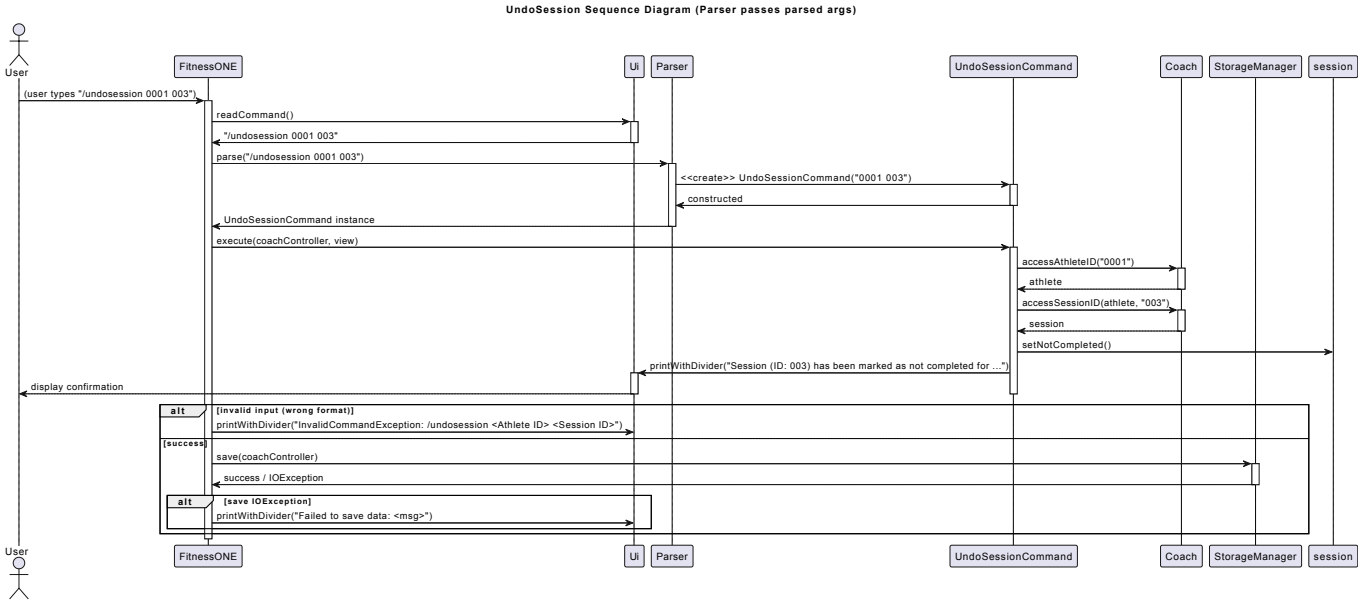
Error handling:

1. If the input format is invalid, UndoExerciseCommand throws InvalidCommandException, and Ui.printWithDivider displays the message.
2. If the athlete, session, or exercise is not found, Coach throws InvalidAthleteException / InvalidSessionException / InvalidExerciseException; the error is caught and displayed via Ui.printWithDivider.
3. FitnessONE.run does not call save on failure.

UndoSession feature

Purpose: Mark a previously completed session as not completed. Implemented by UndoSessionCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
2. FitnessONE passes the raw string to Parser.parse(...), which constructs an UndoSessionCommand instance.
3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to UndoSessionCommand.execute.
4. UndoSessionCommand.execute: it asks Coach for the Athlete (coachController.accessAthleteID) and then for the Session (coachController.accessSessionID), calls session.setNotCompleted() to mark the session as not completed, and prints confirmation via view.printWithDivider.
5. Control returns to FitnessONE.run; run persists state with StorageManager.save(coachController) and reports any I/O errors to the Ui.

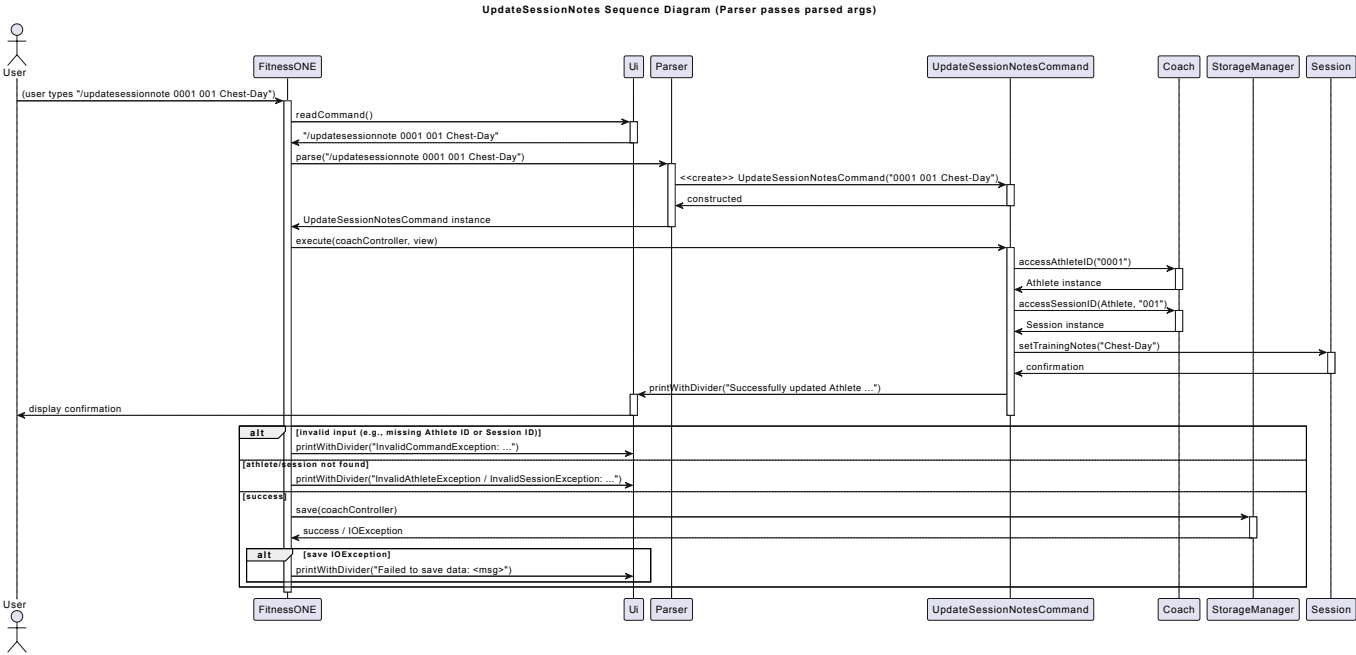
Error handling:

1. If the athlete or session is not found, Coach throws InvalidAthleteException / InvalidSessionException.
2. UndoSessionCommand wraps this into InvalidCommandException (or the error is handled in run()), then Ui.printWithDivider displays the message.
3. FitnessONE.run does not call save on failure.

UpdateSessionNotes feature

Purpose: Show how the application handles a user request to update the training notes of a session, implemented by UpdateSessionNotesCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
2. FitnessONE passes the raw string to Parser.parse(...), which constructs an UpdateSessionNotesCommand instance.
3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to UpdateSessionNotesCommand.execute.
4. UpdateSessionNotesCommand.execute calls coachController.accessAthleteID(athleteID) to retrieve the Athlete, then calls coachController.accessSessionID(athlete, sessionID) to retrieve the Session. The command then calls session.setTrainingNotes(sessionNotes) to update the notes and prints confirmation via view.printWithDivider(...).
5. Control returns to FitnessONE.run; if the command was successful, StorageManager.save(coachController) is invoked to persist changes.

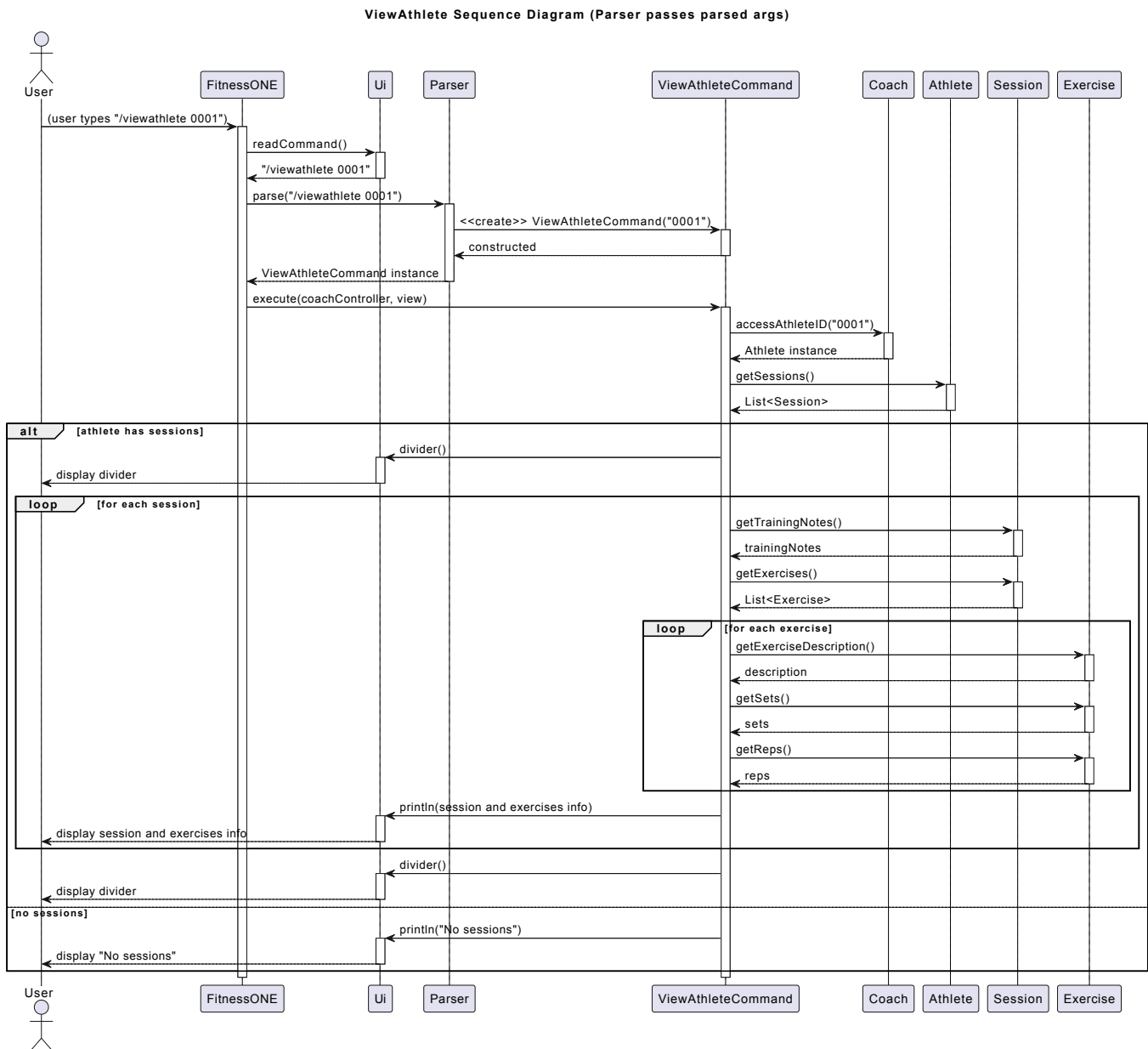
Error handling:

1. If the athlete or session is not found, Coach throws InvalidAthleteException / InvalidSessionException.
2. If the input string is invalid, Parser or command throws InvalidCommandException.
3. Exceptions are wrapped or caught in FitnessONE.run() and displayed via Ui.printWithDivider(...).
4. On failure, StorageManager.save is not called.

ViewAthlete feature

Purpose: show how the application handles a user request to view an athlete’s sessions and exercises, which is implemented by ViewAthleteCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
2. FitnessONE passes the raw string to Parser.parse(...), which constructs a ViewAthleteCommand instance.
3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to ViewAthleteCommand.execute.
4. ViewAthleteCommand.execute: it asks Coach for the Athlete (coachController.accessAthleteID), then calls athlete.getSessions() to retrieve all sessions.
5. The command iterates through each session and prints session.getTrainingNotes(). For each session, it iterates through session.getExercises(), printing exercise.getExerciseDescription(), exercise.getSets(), and exercise.getReps() via view.println(...).
6. Control returns to FitnessONE.run. Since this command is read-only, no state persistence (StorageManager.save) is invoked.

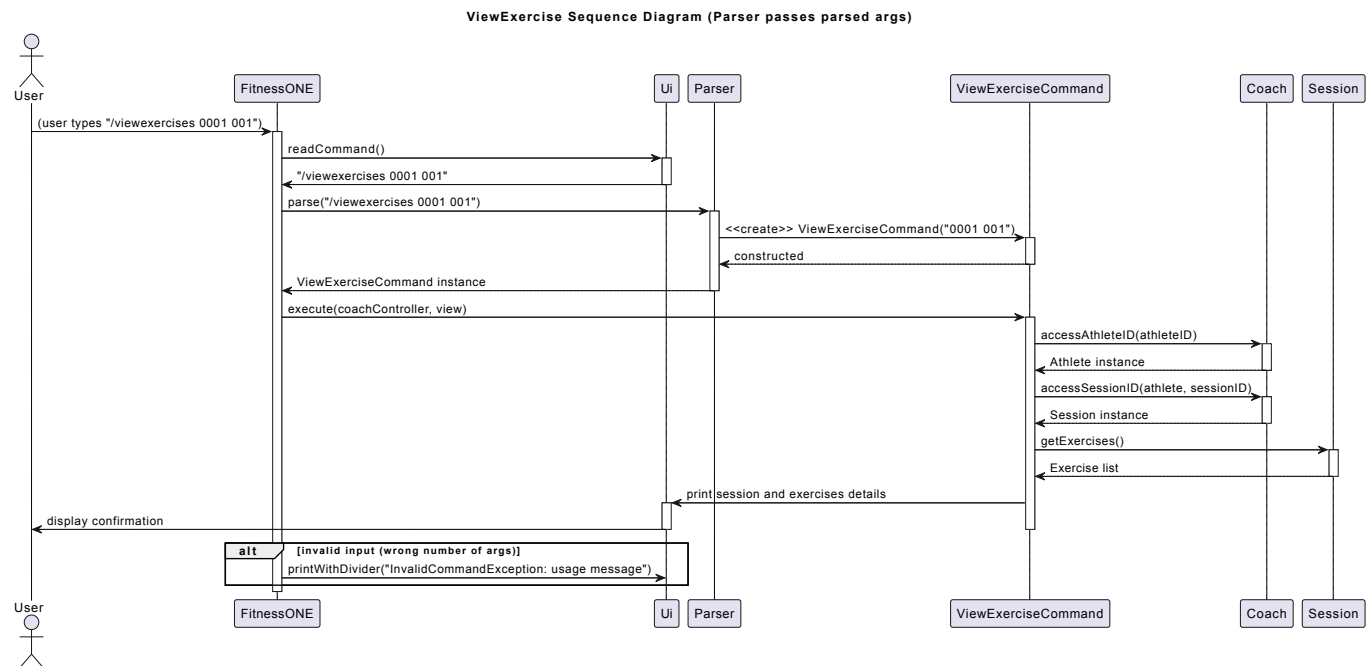
Error handling:

1. If the athlete ID is invalid or not found, Coach throws `InvalidAthleteException`.
2. `ViewAthleteCommand` wraps or propagates this to `InvalidCommandException`, and `Ui.printWithDivider` displays the message.
3. `FitnessONE.run` does not call `save` on failure.

ViewExercise feature

Purpose: show how the application handles a user request to view all exercises in a specified session of an athlete, which is implemented by `ViewExerciseCommand`.

Sequence Diagram



Step-by-step explanation (map to code):

1. `FitnessONE.run` calls `view.readCommand()` to get the raw string (`Ui.readCommand`).
2. `FitnessONE` passes the raw string to `Parser.parse(...)`, which constructs a `ViewExerciseCommand` instance.
3. `FitnessONE` invokes `c.execute(coachController, view)`. This call is statically dispatched by the caller (`FitnessONE`) and dynamically dispatched to `ViewExerciseCommand.execute`.
4. `ViewExerciseCommand.execute`: it asks `Coach` for the `Athlete` (`coachController.accessAthleteID`) and then for the `Session` (`coachController.accessSessionID`).
5. The command iterates over `session.getExercises()` to retrieve each `Exercise`'s information and prints the details via `view.printWithDivider` and `view.println`.
6. Control returns to `FitnessONE.run`; since this command does not modify state, no persistence via `StorageManager.save` is needed.

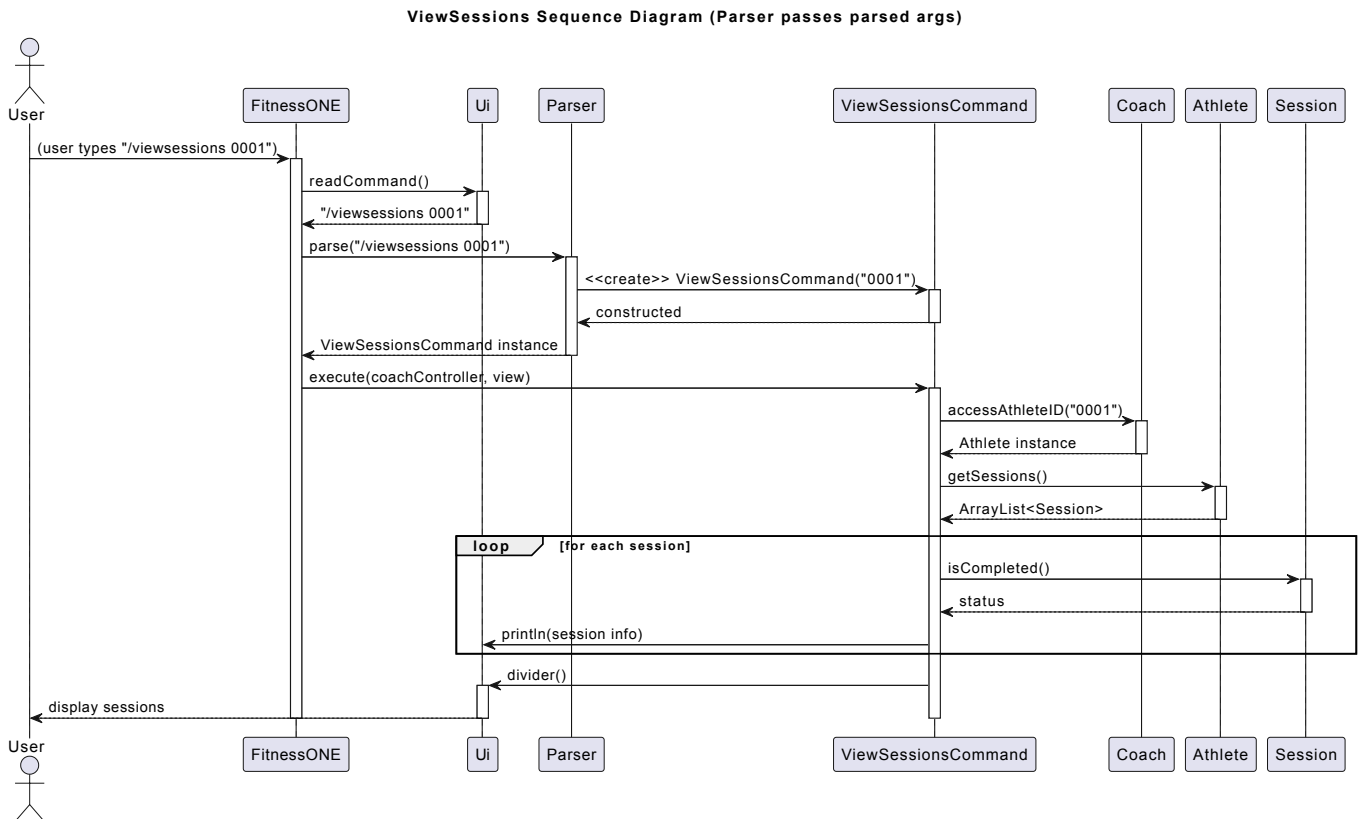
Error handling:

1. If the athlete or session is not found, `Coach` throws `InvalidAthleteException` or `InvalidSessionException`.
2. `ViewExerciseCommand` wraps this into `InvalidCommandException` (or the error is handled in `run()`), then `Ui.printWithDivider` displays the message.
3. `FitnessONE.run` does not call `save` on failure.

ViewSessions feature

Purpose: show how the application handles a user request to view all sessions of a specified athlete, implemented by ViewSessionsCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
2. FitnessONE passes the raw string to Parser.parse(...), which constructs a ViewSessionsCommand instance.
3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to ViewSessionsCommand.execute.
4. ViewSessionsCommand.resolve: it asks Coach for the Athlete (coachController.accessAthleteID) and then retrieves the list of sessions (athlete.getSessions()).
5. The command iterates through the list of sessions, formats the status, session ID, date, and notes, and prints each session via view.println(...).
6. Control returns to FitnessONE.run; run persists state with storage.save(coachController) if applicable and reports any I/O errors to the Ui.

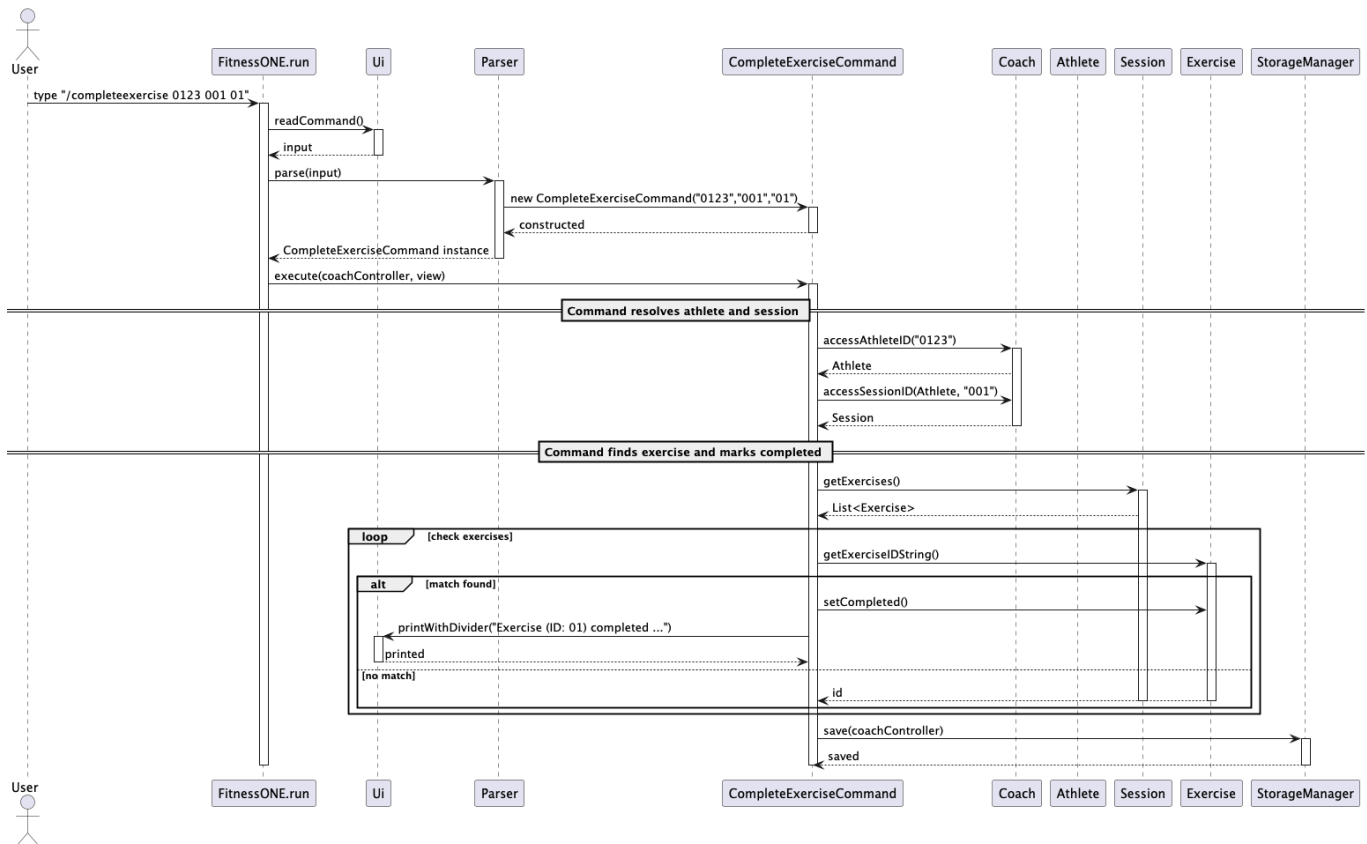
Error handling:

1. If the input string is malformed (too many arguments), ViewSessionsCommand throws InvalidCommandException.
2. If the athlete ID is invalid or the athlete does not exist, Coach throws InvalidAthleteException; the exception is displayed by Ui.printWithDivider.
3. On any failure, FitnessONE.run does not call save.

CompleteExercise feature

Purpose: show how the application handles a user request to complete a single exercise, which is implemented by CompleteExerciseCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
2. FitnessONE passes the raw string to Parser.parse(...), which constructs a CompleteExerciseCommand instance.
3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to CompleteExerciseCommand.execute.
4. CompleteExerciseCommand.resolve: it asks Coach for the Athlete (coachController.accessAthleteID) and then for the Session (coachController.accessSessionID).
5. The command iterates session.getExercises(), compares exercise.getExerciseIDString(), calls exercise.setCompleted() when matched, and prints confirmation via view.printWithDivider.
6. Control returns to FitnessONE.run; run persists state with storage.save(coachController) and reports any I/O errors to the Ui.

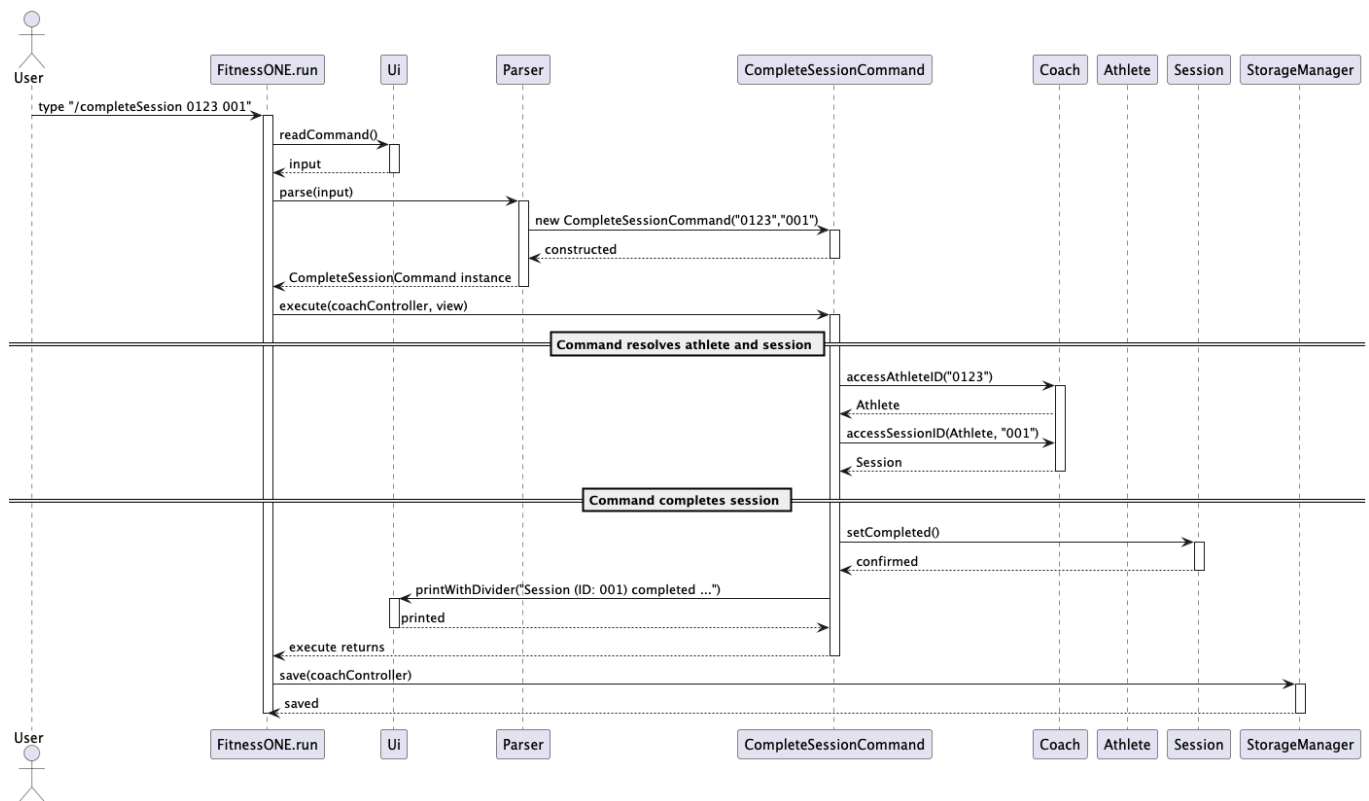
error handling:

1. If the athlete, session or exercise is not found, Coach throws InvalidAthleteException/InvalidSessionException/InvalidExerciseException;
2. CompleteExerciseCommand wraps this into InvalidCommandException (or the error is handled in run()), then Ui.printWithDivider displays the message.
3. FitnessONE.run does not call save on failure.

CompleteSession feature

Purpose: show how the application handles a user request to complete a single session, which is implemented by CompleteSessionCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
2. FitnessONE passes the raw string to Parser.parse(...), which constructs a CompleteSessionCommand instance.
3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to CompleteSessionCommand.execute.
4. CompleteSessionCommand.resolve: it asks Coach for the Athlete (coachController.accessAthleteID) and then for the Session (coachController.accessSessionID).
5. The command calls session.setCompleted() when matched, and prints confirmation via view.printWithDivider.
6. Control returns to FitnessONE.run; run persists state with storage.save(coachController) and reports any I/O errors to the Ui.

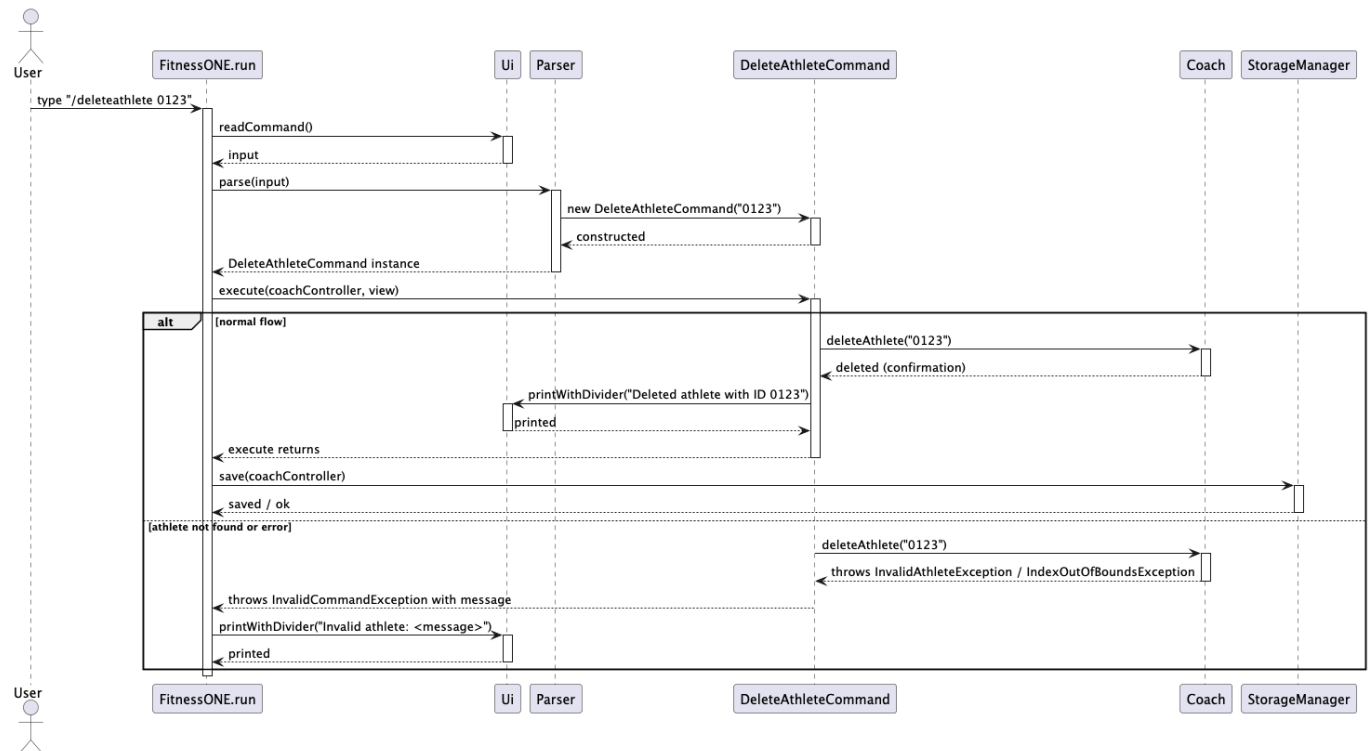
error handling:

1. If the athlete or session is not found, Coach throws InvalidAthleteException/InvalidSessionException;
2. CompleteSessionCommand wraps this into InvalidCommandException (or the error is handled in run()), then Ui.printWithDivider displays the message.
3. FitnessONE.run does not call save on failure.

DeleteAthlete feature

Purpose: show how the application handles a user request to delete an athlete, which is implemented by DeleteAthleteCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls `view.readCommand()` to get the raw string (`Ui.readCommand`).
2. FitnessONE passes the raw string to `Parser.parse(...)`, which constructs a `DeleteAthleteCommand` instance.
3. FitnessONE invokes `c.execute(coachController, view)`. This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to `DeleteAthleteCommand.execute`.
4. `DeleteAthleteCommand.resolve`: it asks Coach to call `deleteAthlete()` when matched, and prints confirmation via `view.printWithDivider`.
5. Control returns to `FitnessONE.run`; run persists state with `storage.save(coachController)` and reports any I/O errors to the `Ui`.

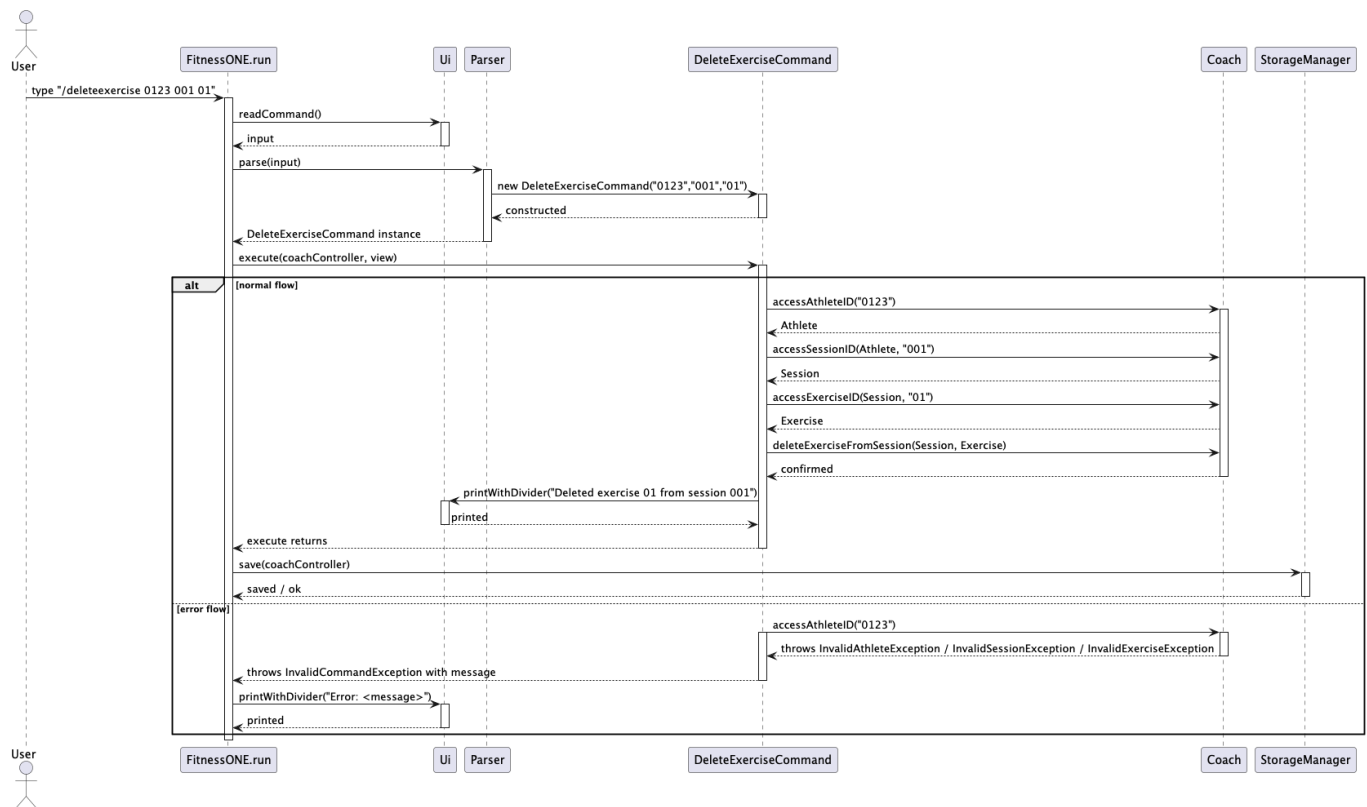
error handling:

1. If the athlete is not found, Coach throws `InvalidAthleteException`;
2. `DeleteAthleteCommand` wraps this into `InvalidCommandException` (or the error is handled in `run()`), then `Ui.printWithDivider` displays the message.
3. `FitnessONE.run` does not call `save` on failure.

DeleteExercise feature

Purpose: show how the application handles a user request to delete a single exercise, which is implemented by `DeleteExerciseCommand`.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls `view.readCommand()` to get the raw string (`Ui.readCommand()`).
2. FitnessONE passes the raw string to `Parser.parse(...)`, which constructs a `DeleteExerciseCommand` instance.
3. FitnessONE invokes `c.execute(coachController, view)`. This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to `DeleteExerciseCommand.execute`.
4. `DeleteExerciseCommand.resolve`: it asks Coach for the Athlete (`coachController.accessAthleteID`) and then for the Session (`coachController.accessSessionID`).
5. The command iterates `session.getExercises()`, compares `exercise.getExerciseIDString()`, calls `DeleteExerciseFromSession()` when matched, and prints confirmation via `view.printWithDivider`.
6. Control returns to `FitnessONE.run`; run persists state with `storage.save(coachController)` and reports any I/O errors to the `Ui`.

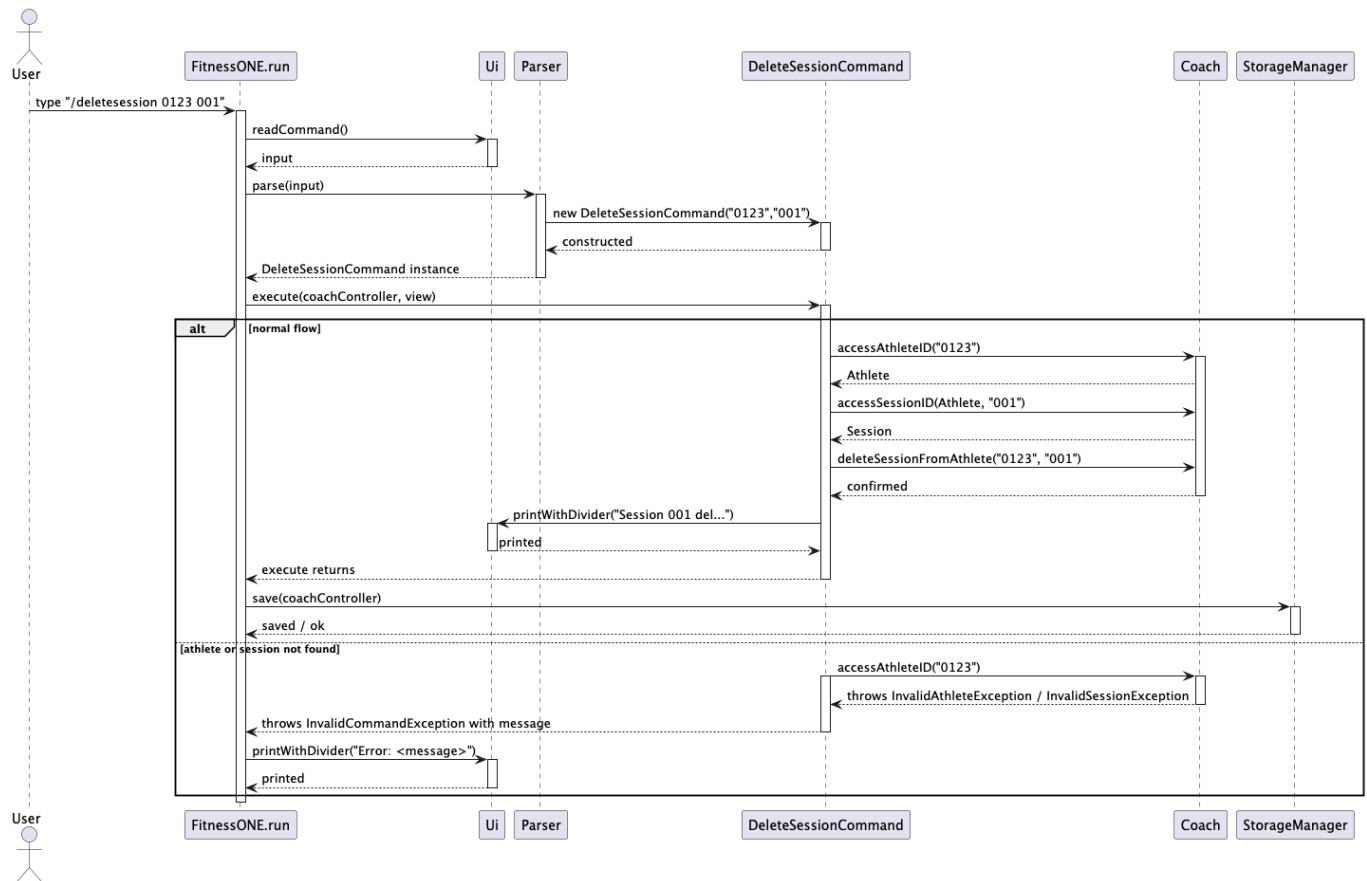
error handling:

1. If the athlete, session or exercise is not found, Coach throws `InvalidAthleteException/InvalidSessionException/InvalidExerciseException`;
2. `DeleteExerciseCommand` wraps this into `InvalidCommandException` (or the error is handled in `run()`), then `Ui.printWithDivider` displays the message.
3. `FitnessONE.run` does not call `save` on failure.

DeleteSession feature

Purpose: show how the application handles a user request to delete a single session, which is implemented by `DeleteSessionCommand`.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls `view.readCommand()` to get the raw string (`Ui.readCommand`).
2. FitnessONE passes the raw string to `Parser.parse(...)`, which constructs a `DeleteSessionCommand` instance.
3. FitnessONE invokes `c.execute(coachController, view)`. This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to `DeleteSessionCommand.execute`.
4. `DeleteSessionCommand.resolve`: it asks Coach for the Athlete (`coachController.accessAthleteID`) and then for the Session (`coachController.accessSessionID`).
5. The command calls `coach.deleteSessionFromAthlete()` when matched, and prints confirmation via `view.printWithDivider`.
6. Control returns to `FitnessONE.run`; run persists state with `storage.save(coachController)` and reports any I/O errors to the `Ui`.

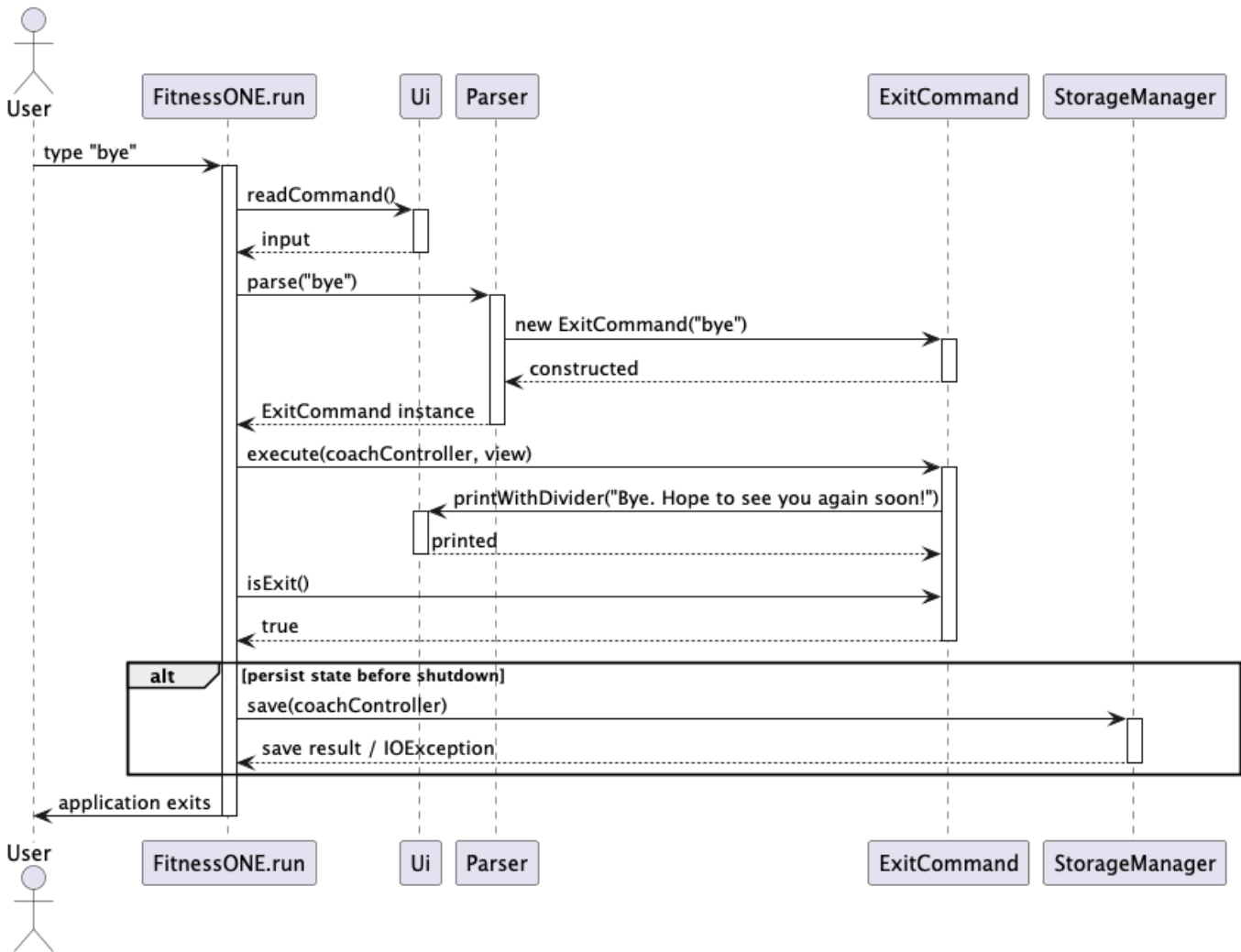
error handling:

1. If the athlete or session is not found, Coach throws `InvalidAthleteException/InvalidSessionException`;
2. `DeleteSessionCommand` wraps this into `InvalidCommandException` (or the error is handled in `run()`), then `Ui.printWithDivider` displays the message.
3. `FitnessONE.run` does not call `save` on failure.

Exit feature

Purpose: show how the application handles a user request to exit, which is implemented by `ExitCommand`.

Sequence Diagram



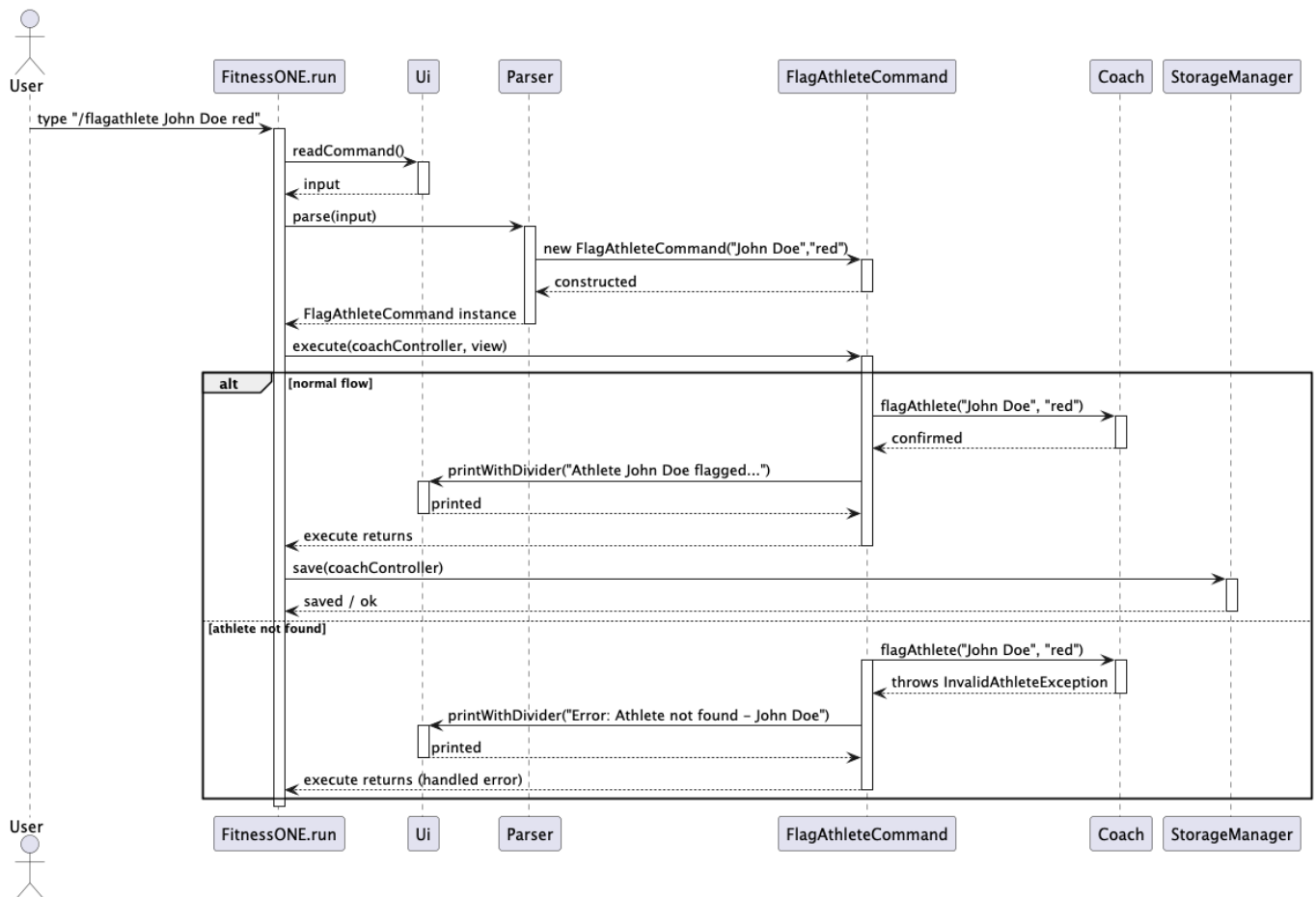
Step-by-step explanation (map to code):

- 1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
- 2. FitnessONE passes the raw string to Parser.parse(...), which constructs a ExitCommand instance.
- 3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to ExitCommand.execute.
- 4. The command prints confirmation via view.printWithDivider.
- 5. Control returns to FitnessONE.run; run persists state with storage.save(coachController) and reports any I/O errors to the Ui.

FlagAthlete feature

Purpose: show how the application handles a user request to flag an athlete, which is implemented by FlagAthleteCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls `view.readCommand()` to get the raw string (`Ui.readCommand()`).
2. FitnessONE passes the raw string to `Parser.parse(...)`, which constructs a `FlagAthleteCommand` instance.
3. FitnessONE invokes `c.execute(coachController, view)`. This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to `FlagAthleteCommand.execute`.
4. `FlagAthleteCommand.resolve`: it asks Coach to call `flagAthlete()` when matched, and prints confirmation via `view.printWithDivider`.
5. Control returns to `FitnessONE.run`; run persists state with `storage.save(coachController)` and reports any I/O errors to the `Ui`.

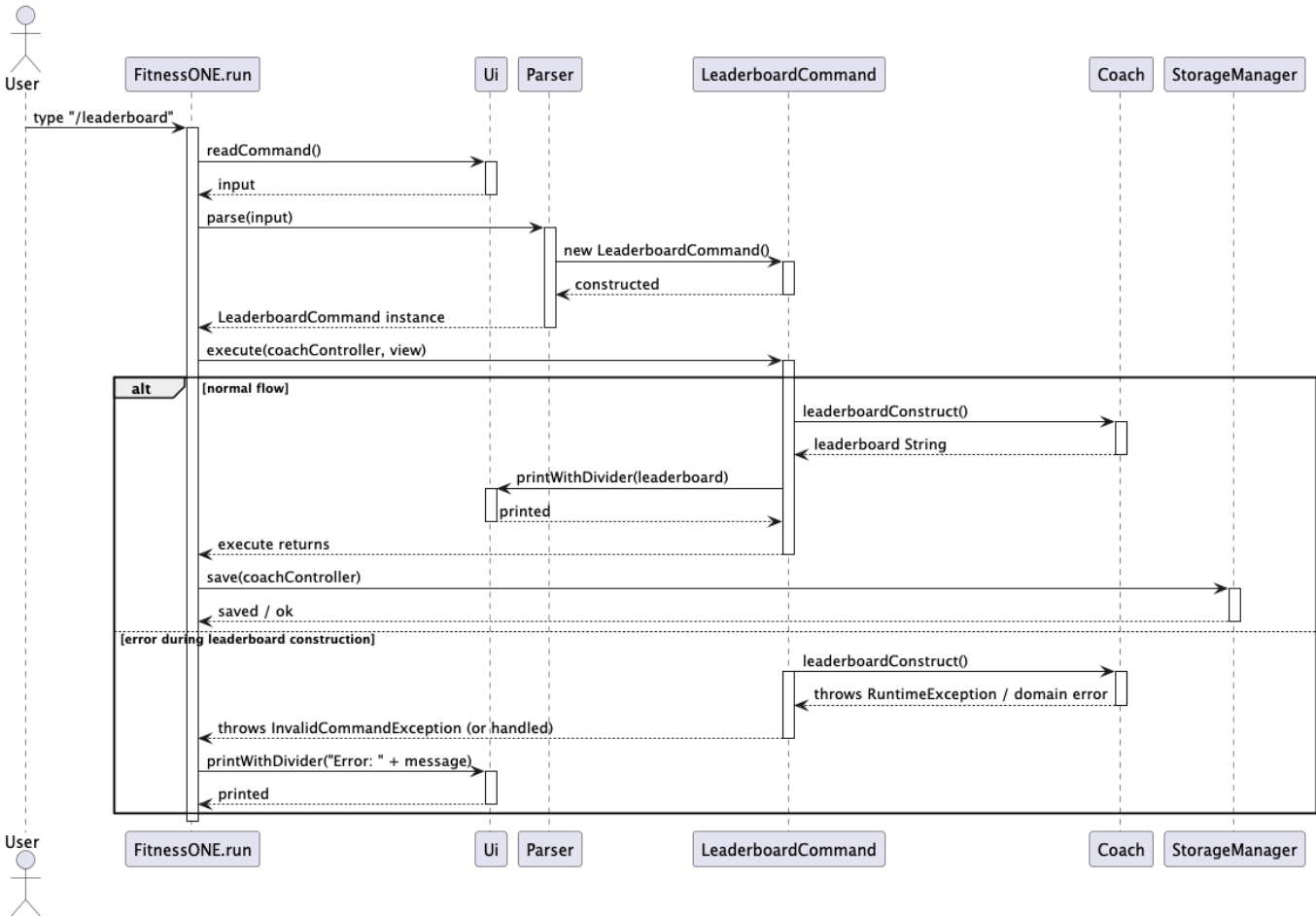
error handling:

1. If the athlete is not found, Coach throws `InvalidAthleteException`;
2. `FlagAthleteCommand` wraps this into `InvalidCommandException` (or the error is handled in `run()`), then `Ui.printWithDivider` displays the message.
3. `FitnessONE.run` does not call `save` on failure.

Leaderboard feature

Purpose: show how the application handles a user request to show leaderboard, which is implemented by `LeaderboardCommand`.

Sequence Diagram



Step-by-step explanation (map to code):

- 1. FitnessONE.run calls view.readCommand() to get the raw string (Ui.readCommand).
- 2. FitnessONE passes the raw string to Parser.parse(...), which constructs a LeaderboardCommand instance.
- 3. FitnessONE invokes c.execute(coachController, view). This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to LeaderboardCommand.execute.
- 4. LeaderboardCommand.resolve: it asks Coach to call leaderboardConstruct(), and prints leaderboard via view.printWithDivider.
- 5. Control returns to FitnessONE.run; run persists state with storage.save(coachController) and reports any I/O errors to the Ui.

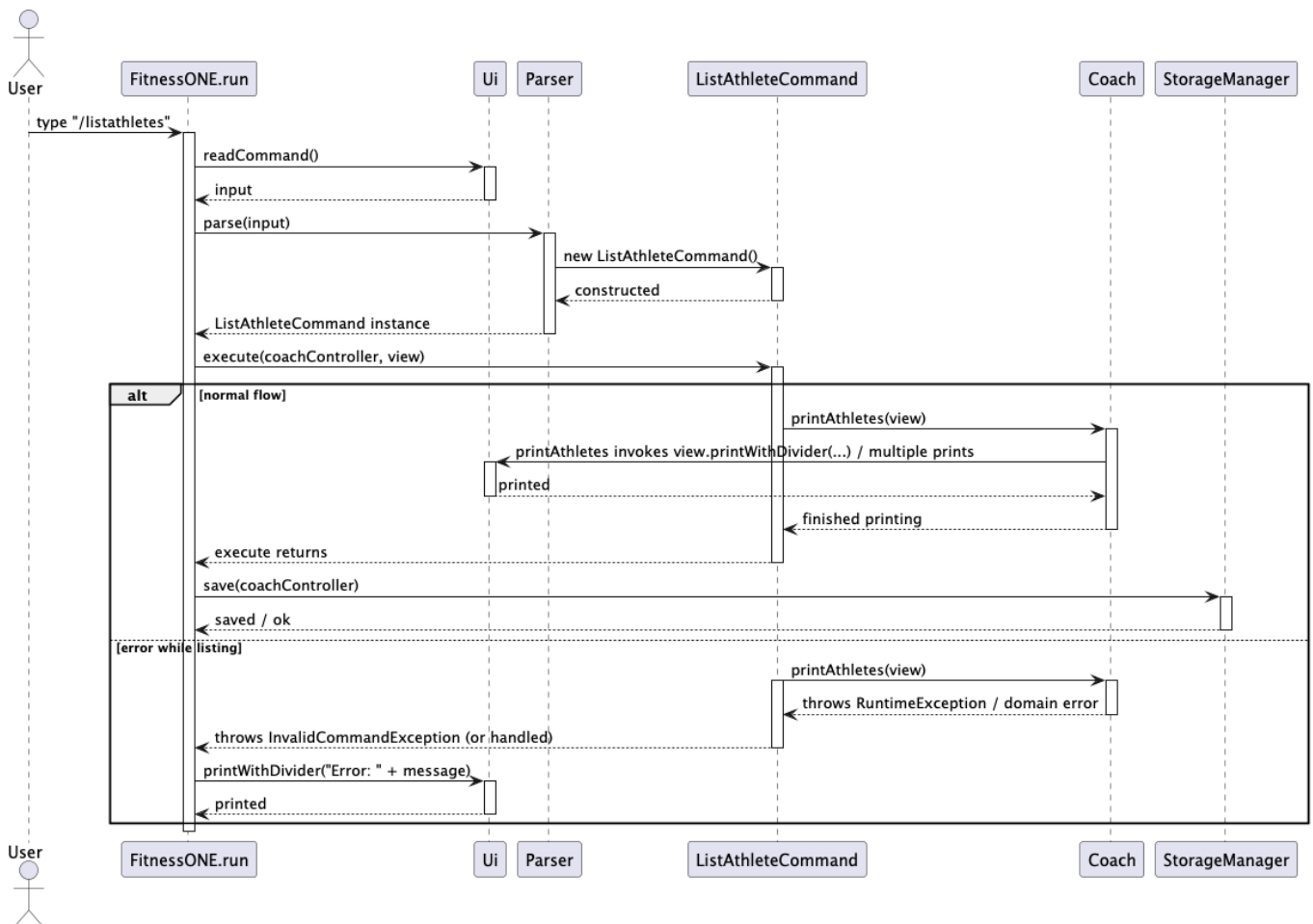
error handling:

- 1. If the input contains redundant message, LeaderboardCommand throws InvalidCommandException (or the error is handled in run()), then Ui.printWithDivider displays the message.
- 2. FitnessONE.run does not call save on failure.

ListAthlete feature

Purpose: show how the application handles a user request to list athletes, which is implemented by ListAthleteCommand.

Sequence Diagram



Step-by-step explanation (map to code):

1. FitnessONE.run calls `view.readCommand()` to get the raw string (`Ui.readCommand()`).
2. FitnessONE passes the raw string to `Parser.parse(...)`, which constructs a `ListAthleteCommand` instance.
3. FitnessONE invokes `c.execute(coachController, view)`. This call is statically dispatched by the caller (FitnessONE) and dynamically dispatched to `ListAthleteCommand.execute`.
4. `printAthletes.resolve`: it asks Coach to call `printAthlete()`, and prints athletes via `view.printWithDivider`.
5. Control returns to `FitnessONE.run`; run persists state with `storage.save(coachController)` and reports any I/O errors to the `Ui`.

error handling:

1. If the input contains redundant message, `ListAthleteCommand` throws `InvalidCommandException` (or the error is handled in `run()`), then `Ui.printWithDivider` displays the message.
2. `FitnessONE.run` does not call `save` on failure.

Testing

Testing will ensure the correctness, reliability, and stability of `FitnessOne`. The project uses JUnit (version 5) for automated testing and manual verification of the user-facing behavior.

Our methods combine:

- Unit testing for core logic (e.g., `Parser`, `Coach`, `Command` classes).

- Integration testing for features that involve multiple components.
- System testing for end-to-end behavior using the CLI interface. Running Tests

Tests can be automatically executed using Gradle:

```
./gradlew clean test
```

The build itself will fail if any test fails. In order to open a full report use:

```
build/reports/tests/test/index.html
```

you can also access specific test cases:

```
./gradlew test --tests seedu.fitnessone.command.NewSessionCommandTest
```

Tests

For the tests themselves we split them up first of all in different components to ensure as full coverage as possible.

- Unit tests
 - Model tests (AthleteTest, SessionTest, ExerciseTest) to ensure data integrity and correct state updating.
 - Controller tests (CoachTest) which verify coordination between model objects and correct exception propagation.
 - Command tests (NewAthleteCommandTest, DeleteSessionCommandTest...): to check user input parsing, command execution, and outputs.
 - Parser test: confirming valid/invalid command strings if they are correctly interpreted which are vital to the correct functionality.
- Integration test, to verify collaboration between logic and model layers. Thus ensuring data consistency across components:
 - Creating a new athlete and session (NewAthleteCommand + NewSessionCommand).
 - Adding and deleting exercises via the Coach controller.
- System Tests (CLI), testing the full command lifecycle, from user input to UI output.
 - System tests simulate user workflows end-to-end: Adding an athlete -> adding a session -> adding an exercise -> deleting it
 - Handling invalid inputs and ensuring helpful error messages are displayed

Appendix: Instructions for Manual Testing

Launch and shutdown

This section will assist and guide testers through manual validation of FitnessOne's core features. Make sure that the JAR file has been correctly built (e.g., FitnessOne.jar) and can be run in a terminal:

```
java -jar FitnessOne.jar
```

To shutdown the program simply type the command: **bye**

this will lead to the program terminating.

Notes

Copy and paste the commands one by one to verify command parsing and UI responses are correct. You do not need to test every invalid variation, make sure at least one invalid example per command behaves correctly.

1. Adding an Athlete

Test Case: **/newAthlete Jonas Hardwell**

Expected: **New athlete created: Jonas Hardwell**

Notes: Name cannot be blank.

Error Case: **/newAthlete => athlete name was not specified**

2. Adding a Training Session

Prerequisite: An athlete exists (e.g., John Doe with ID 0001).

Test Case: **newsession 0001 Legs**

Expected:

```
New session created:  
Athlete Name: jonas hardwell | ID: 0001  
  
Session ID: 002  
Session Description: legs
```

Error Case: **/newsession 9999 Legs => Athlete not found: 9999**

3. Adding an Exercise

Prerequisite: The Athlete and session exist.

Test Case: **/newExercise 0001 001 Pushups 3 12**

Expected:

```
New exercise created!

Athlete (ID) : 0001
Athlete name: jonas hardwell

Session (ID): 001
Session Description: chest

Exercise (ID): 03
Exercise Description: leg-press
sets x reps: 5 x 15
```

Error Case: `newExercise 0001 001 Pushups five 12` => Sets and reps must be integers.

4. Mark an Exercise Complete

Prerequisite: Exercise 01 exists for Session 001, which exists for athlete 0001.

Test Case: `/completeExercise 0001 001 01`

Expected: `Exercise (ID: 01) completed by Jonas Hardwell (ID: 0001).`

Error Case: `/completeExercise 0001 001 99` => There was an error while trying to complete the session. Try: `/completeSession <ATHLETE_ID> <SESSION_ID> <EXERCISE_ID>`

5. Mark a Session Complete

Prerequisite: Session 001 exists for athlete 0001.

Test Case: `/completeSession 0001 001`

Expected: `Session (ID: 001) completed by Jonas Hardwell (ID: 0001).`

Error Case: `/completeSession 0001 999` => "There was an error while trying to complete the session. Try: `/completeSession <ATHLETE_ID> <SESSION_ID>`"

6. Unmark an Exercise Complete

Prerequisite: Exercise 01 exists for Session 001, which exists for athlete 0001.

Test Case: `/undoExercise 0001 001 01`

Expected: `Exercise (ID: 01) has been marked as not completed by Jonas Hardwell (ID: 0001).`

Error Case: `/undoExercise 0001 001 99` => Exercise not found: 99

7. Unmark a Session Complete

Prerequisite: Session 001 exists for athlete 0001.

Test Case: `/undoSession 0001 001`

Expected: Session (ID: 001) has been marked as not completed by Jonas Hardwell (ID: 0001).

Error Case: `/undoSession 0001 999 => Session not found: 999`

6. Deleting an Exercise

Prerequisite: Exercise 01 exists for Session 001, which exists for athlete 0001.

Test Case: `/deleteExercise 0001 001 01`

Expected: Exercise (ID: 01), for Session (ID: 001) deleted for Jonas Hardwell (ID: 0001)

Error Case: `/deleteExercise 0001 001 99 => Exercise not found: 99`

7. Deleting a Session

Prerequisite: Session 001 exists for athlete 0001.

Test Case: `/deleteSession 0001 001`

Expected: Session (ID: 003) deleted for Jonas Hardwell (ID: 0001)

Error Case: `/deleteSession 0001 999 => Session not found: 999`

8. Flagging an Athlete

Prerequisite: athlete 0001 exists.

Test Case: `/flagathlete 0001 Red`

Expected: Athlete 0001 flagged as: Red

Error Case 1: `/flagathlete 9999 Red => Athlete not found: 9999`

Error Case 2: `/flagathlete 0001 reddd => Invalid color: reddd`

9. Deleting an Athlete

Test Case: `/deleteathlete 0001`

Expected: Deleted athlete with ID 0001

Error Case: `/deleteathlete 9999 => Athlete not found: 9999`

10. Viewing the Leaderboard

Test Case: `/leaderboard`

Expected: A list of athletes sorted by achievement score.

If empty: **No athletes found, add some athletes and let them do workout!!**

11. Error Handling

Test invalid commands:

```
/unknowncommand
```

Expected:

```
Invalid Command. Type /help for a list of available commands.
```

Instructions for Manual Testing (storage)

This section provides a short path a tester can follow to verify the storage feature (startup load and save-on-command).

1. Start with a fresh/known state:

- If you already have a **data/athletes_export.txt** file, back it up or delete it to observe creation from scratch.

2. Run the application (from project root):

```
./gradlew run
```

3. Verify startup behaviour:

- With no **data/athletes_export.txt**: observe the welcome message and the printed fallback message ("No saved data found, starting with empty data").
- With an existing file containing saved data: the app should report "Loaded saved athletes data." and the athletes/sessions/exercises should be accessible via view commands.

4. Test save-on-command:

- Add an athlete using the appropriate command (see User Guide). For example (copy-paste):
 - **addAthlete A01 John Doe** (replace with actual command used in the app)
- Add a session/exercise for that athlete.
- Exit the application using the **exit** command.
- Confirm that **data/athletes_export.txt** was created/updated and contains lines starting with **ATHLETE|**, **SESSION|**, **EXERCISE|**.

5. Test load-on-startup round-trip:

- Restart the application.
- Confirm the previously added athlete and sessions are present.

6. Edge checks:

- Make sure multi-line notes are preserved across save/load (newlines are escaped as `\n` on save and restored on load).
- Confirm that saving overwrites the data file (it does not append duplicate entries for the same runtime state).

If any of the above steps fail, capture the console output and the contents of `data/athletes_export.txt` and file an issue with those artifacts.

Product scope

Target User profile:

FitnessONE is designed for professional fitness coaches and personal trainers who:

- Manage multiple athletes simultaneously
- Need to track detailed training programs, including sessions and individual exercises
- Can type fast and are comfortable using command-line interfaces
- Value speed, data accuracy, and organization in their workflow
- Want to monitor athlete progress through completion tracking and performance metrics
- Prefer quick access to athlete information without navigating through multiple GUI screens
- Need to prioritize athletes based on urgency or training status

Value Proposition:

FitnessONE empowers coaches to manage and monitor athletes’ progress more efficiently than traditional GUI-based fitness management software by:

- Allowing fast creation, viewing, and management of athletes, sessions, and exercises directly from the command line
- Automating ID assignment and progress tracking for each athlete, session, and exercise
- Providing clear, structured summaries of training data and athlete performance
- Including a leaderboard feature that helps coaches compare athletes’ progress and use insights to motivate their teams
- Offering visual prioritization through color-coded athlete flags (red, yellow, green, blue, ...) for status or urgency tracking
- Storing all information locally to ensure data privacy and offline availability
- Automatically saving all changes to eliminate the need for manual save operations

User Stories

Version	As a...	I want to...	So that I can...
v1.0	Fitness coach	add a new athlete	start tracking their sessions and performance individually
v1.0	Fitness coach	view athlete details	see all sessions and exercises for a specific athlete

Version	As a...	I want to...	So that I can...
v1.0	Fitness coach	list all athletes	get an overview of everyone I'm coaching
v1.0	Fitness coach	delete an athlete	remove clients who are no longer active
v1.0	Fitness coach	create a new session	plan structured workouts for an athlete
v1.0	Fitness coach	delete a session	remove outdated or incorrect training sessions
v1.0	Fitness coach	complete a session	mark training sessions as done to track athlete progress
v1.0	Fitness coach	view all sessions	review past and upcoming workouts for an athlete
v1.0	Fitness coach	update session notes	adjust or add relevant details after a session
v1.0	Fitness coach	undo session completion	correct mistakes when marking sessions as completed
v1.0	Fitness coach	create an exercise	define specific workouts within a session
v1.0	Fitness coach	delete an exercise	remove invalid or outdated exercises from a session
v1.0	Fitness coach	view all exercises	review what an athlete must perform in a given session
v1.0	Fitness coach	complete an exercise	track the completion status of each workout
v1.0	Fitness coach	undo exercise completion	revert accidental completions for accuracy
v2.0	Fitness coach	view the leaderboard	compare athletes' performance and identify top performers
v2.0	Fitness coach	view the help menu	understand how to use all available commands
v2.0	Fitness coach	flag athletes	prioritize athletes based on training urgency or performance level
v2.0	Fitness coach	save all athlete data, sessions, and exercises	persist my athletes' progress and training plans for later use
v2.0	Fitness coach	load saved athlete data, sessions, and exercises	restore all information without re-entering it manually

Version	As a...	I want to...	So that I can...
v2.1	Fitness coach	run the app smoothly without any major bugs	can use the app properly

Use cases

For all use cases below, the System is FitnessONE and the Actor is the user, unless otherwise specified.

Use Case: Add a New Athlete

MSS

1. User enters the command to add a new athlete (e.g., /newathlete John Doe)
2. System creates a new athlete profile, assigns a unique ID, and adds it to storage
3. System confirms with a message displaying the newly assigned ID

Extensions 2a. Athlete name is empty or invalid 2a1. FitnessOne shows an error message

Use Case: Delete an Athlete

MSS

1. User issues the command to delete an athlete (e.g., /deleteathlete 0001)
2. System checks if the athlete exists
3. System deletes the athlete and all associated sessions and exercises
4. System confirms the deletion with a message

Extensions

2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message

Use Case: Add a New Session

MSS

1. User enters the command to add a session (e.g., /newsession 0001 Leg day)
2. System checks if the athlete ID exists
3. System creates a session, assigns a unique session ID, and adds it to the athlete profile
4. System confirms the new session with details (ID, notes)

Extensions

2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message

3a. Sessions Note is empty or invalid 3a1. FitnessOne shows an error message

Use Case: Delete a Session

MSS

1. User issues delete session command (e.g., /deletesession 0001 001)

2. System checks athlete and session IDs
3. System deletes the session and all its exercises
4. System confirms the session deletion

Extensions

- 2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 2a. Session ID invalid, empty or not found 2a1. FitnessOne shows an error message

Add an Exercise

MSS

1. User commands to add an exercise (e.g., /newexercise 0001 001 Bench Press 5 10)
2. System validates athlete and session IDs
3. System adds the exercise to the session with a unique ID
4. System confirms creation with exercise details

Extensions

- 2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 2a. Session ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 3a. Invalid command format 3a1. FitnessOne shows an error message

Use Case: Delete an Exercise

MSS

1. User enters the delete exercise command (e.g., /deleteexercise 0001 001 01)
2. System checks if athlete, session, and exercise IDs exist
3. System deletes the specified exercise
4. System confirms deletion to the user

Extensions

- 2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 2a. Session ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 3a. Exercise ID invalid, empty or not found 3a1. FitnessOne shows an error message

Use Case: Complete an Exercise

MSS

1. User enters the complete exercise command (/completeexercise 0001 001 01)
2. System verifies that all IDs are valid and exist
3. System marks the exercise as completed
4. System confirms completion to the user

Extensions

- 2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 2a. Session ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 3a. Exercise ID invalid, empty or not found 3a1. FitnessOne shows an error message

Use Case: Complete a Session

MSS

1. User enters the complete session command (/completesession 0001 001)
2. System verifies that all IDs (session and athlete ID) are valid and exist
3. System marks the session as completed
4. System confirms completion to the user

Extensions

- 2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 2a. Session ID invalid, empty or not found 2a1. FitnessOne shows an error message

Use Case: Undo an Exercise

MSS

1. User enters the complete exercise command (undoexercise 0001 001 01)
2. System verifies that all IDs are valid and exist
3. System marks the exercise as not completed
4. System confirms completion to the user

Extensions

- 2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 2a. Session ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 3a. Exercise ID invalid, empty or not found 3a1. FitnessOne shows an error message

Use Case: Complete a Session

MSS

1. User enters the complete session command (/undosession 0001 001)
2. System verifies that all IDs (session and athlete ID) are valid and exist
3. System marks the session as completed
4. System confirms completion to the user

Extensions

- 2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message

2a. Session ID invalid, empty or not found 2a1. FitnessOne shows an error message

Use Case: Flag an Athlete

MSS

1. User enters the command to flag an athlete (e.g., /flagathlete 0001 red)
2. System checks athlete ID, validates color
3. System updates the flag color
4. System confirms flag update

Extensions

2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message

3a. Flag color invalid, empty 3a1. FitnessOne shows an error message

Use Case: List All Athletes

MSS

1. User enters the command to list all athletes (/listathletes)
2. System retrieves a list of all athletes, including flags and IDs
3. System displays the list in a clear format to the user

Use Case: View Athlete Details

MSS

1. User enters the command to view an athlete (e.g., /viewathlete 0001)
2. System checks if the athlete ID exists
3. System retrieves and displays all details for that athlete, including sessions and exercises

Extensions

2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message

Use Case: View All Sessions for an Athlete

MSS

1. User enters the command to view all sessions for an athlete User enters the command to view all exercises (e.g., /viewexercises 0001 001)
2. System verifies the athlete ID
3. System retrieves and displays all sessions for the specified athlete

Extensions

2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message

Use Case: View All Exercises in a Session

MSS

1. User enters the command to view all exercises (e.g., /viewexercises 0001 001)
2. System checks both athlete and session IDs
3. System retrieves and displays all exercises in the specified session

Extensions

- 2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 2a. Session ID invalid, empty or not found 2a1. FitnessOne shows an error message

Use Case: Update Session Notes

MSS

1. User enters the command to update session notes (e.g., /updatesessionnote 0001 001 push)
2. System checks athlete and session IDs
3. System updates the session's notes with the new text
4. System confirms update with a message

Extensions

- 2a. Athlete ID invalid, empty or not found 2a1. FitnessOne shows an error message
- 2a. Session ID invalid, empty or not found 2a1. FitnessOne shows an error message

Use Case: View Help (List All Commands)

MSS

1. User enters /help
2. System displays a list of all available commands with descriptions and usage

Use Case: View Leaderboard

MSS

1. User enters /leaderboard
2. System calculates athlete scores based on completion (sessions, exercises)
3. System sorts athletes by score and displays the ranking

Use Case: Persist and Restore Coach Data

MSS

1. The user executes actions that modify the coach's data (e.g. adding athletes, sessions, exercises)
2. The System (FitnessONE) calls StorageManager's save() method to persist the data to file
3. Data is written in a structured, line-based format
4. When the application starts (or user requests a reload), the System calls StorageManager's load() method

5. StorageManager parses the file and reconstructs all Coach, Athlete, Session, and Exercise objects in memory, restoring the full application state

Extensions

- 2a. The data file does not exist yet 2a1. StorageManager creates any necessary folders and returns an empty Coach
- 2b. The file exists but is empty 2b1. StorageManager loads an empty Coach
- 4a. The file is malformed (incorrect ID length/type, missing fields, parsing error) 4a1. StorageManager records all parse errors, displays them (stderr), and attempts to recover or skip problematic lines
- 4b. Some IDs are duplicated or invalid 4b1. StorageManager skips faulty records and proceeds with valid ones only
- 5a. The final loaded data is inconsistent (e.g., missing athlete/session/exercise IDs) 5a1. StorageManager asserts and logs errors, but the app remains operational if possible

Non-Functional Requirements

1. Performance

- FitnessONE should respond to user commands within **few** under normal usage
- The system should handle up to **9999 athletes, 999 sessions per athletes, and 99 exercises per session**
- File read/write operations (saving and loading) should complete within **few seconds**

2. Reliability

- Data should be automatically saved after every major operation (e.g., adding/deleting athletes, sessions, exercises)

3. Compatibility

- FitnessONE must run on any operating system with **Java 17 or higher** installed.
- It should work consistently across **Windows, macOS, and Linux**.

4. Usability

- Commands should follow a consistent CLI syntax and argument order.
- Error messages must clearly explain what went wrong and how to fix it.
- Leaderboards and progress summaries should be easy to read and interpret quickly.

5. Maintainability

- The codebase must remain modular — each component (UI, Logic, Model, Storage) should be independently testable.
- All public classes and methods must include concise Javadoc comments describing their purpose and behavior.

6. Extensibility

- New commands or data types should be implementable without modifying core logic, following the Command pattern.
- The modular architecture should allow easy integration of new features such as export, or performance analytics.

7. Security

- File operations should be restricted to FitnessONE's working directory to prevent unauthorized access.
- Deserialization must reject malformed or unexpected input to ensure application safety.

8. Portability

- The application must be distributable as a single standalone `.jar` file with no external dependencies.
- All file paths should be relative, ensuring consistent behavior across machines.

9. Testing Coverage

- Each component should achieve at least **90% unit test coverage**.
- Core features (Add/Delete athletes, sessions, exercises, leaderboard) must include valid JUnit test cases.

Glossary

CLI (Command-Line Interface)

A text-based interface where users interact with FitnessONE by typing commands, instead of using a graphical interface. The CLI allows fast, efficient management of athletes, sessions, and exercises.

Coach / Fitness Coach

The primary user of FitnessONE. Coaches create, manage, and track athletes, their sessions, and exercises. They also monitor performance using leaderboards and flagging.

Athlete

An individual whose training is tracked within FitnessONE. Each athlete has a unique ID, a list of sessions, exercises, notes, and can be flagged for priority tracking.

Session

A structured workout or training period assigned to an athlete. Sessions contain exercises, notes, completion status, and can be added, updated, completed, or deleted.

Exercise

A specific activity within a session. Exercises have details like name, sets, reps, and completion status. They can be added, viewed, completed, undone, or deleted.

Leaderboard

A ranking of athletes based on performance metrics, such as completed sessions and exercises. Coaches can view leaderboards to monitor progress and motivate athletes.

Flagging

A visual priority system where athletes are assigned colors (e.g., red, yellow, green, blue) to indicate training urgency or performance level.

Persistence / Save & Load

Functionality that allows all athletes, sessions, and exercises to be saved to a file and later restored. This ensures data continuity and eliminates the need to re-enter information.

Command

A specific instruction entered by the user via the CLI to perform actions like adding an athlete, completing an exercise, or viewing the leaderboard.

ID (Identifier)

A unique code assigned to each athlete, session, and exercise. IDs ensure accurate referencing and tracking across the system.

Completion / Undo

Actions that allow marking sessions or exercises as completed, and optionally undoing them to correct mistakes.

Help Menu

A CLI feature that lists all available commands and their usage, helping users understand how to interact with the system efficiently.

StorageManager

The component responsible for persisting and restoring data. It handles saving/loading athlete profiles, sessions, exercises, and leaderboard information, ensuring data integrity.