Overview of the `DataStorage` Class

**Package:** `com.data_management`

**Purpose:**

The `DataStorage` class serves as the central repository for managing and retrieving patient data in the cardiovascular data simulator. It organizes patient records by patient ID, storing `Patient` objects in a `HashMap` and providing methods to add data, retrieve records within a time range, and access all patients. The class acts as a bridge between data generation (from `com.cardio_generator.generators`) and data processing (e.g., alert generation in `com.alerts`), enabling structured storage and access to health metrics like heart rate, blood pressure, and ECG.

**Role in the Project:**

The `DataStorage` class is the backbone of the simulator's data management subsystem, coordinating the storage of data produced by `PatientDataGenerator` implementations (e.g., `ECGDataGenerator`, `BloodPressureDataGenerator`) and facilitating data retrieval for analysis or monitoring. It interacts with the `Patient` and `PatientRecord` classes to manage patient-specific records and supports Task 7's alert system by providing data to `com.alerts.AlertGenerator` for evaluation, which may trigger `Alert` objects via `AlertFactory` and `AlertDecorator`. The class is used by `HealthDataSimulator` to store data output via `OutputStrategy` implementations (e.g., `FileOutputStrategy`, `WebSocketOutputStrategy`) and supports both real-time and historical data access, critical for clinical monitoring and alert generation.

**Technical Characteristics:**

- The class uses a `HashMap` to store `Patient` objects indexed by `patientId`, enabling O(1) access to patient data.

- It provides methods for adding data (`addPatientData`), retrieving records (`getRecords`), and accessing all patients (`getAllPatients`), supporting CRUD-like operations.

- The class includes a `main` method for demonstration, though it references an undefined `DataReader`, indicating integration with external data sources.

- It is designed for concurrent use in `HealthDataSimulator`'s multi-threaded environment but lacks explicit thread safety, requiring careful handling in concurrent scenarios.

---

#### Internal Components and Their Purposes

The `DataStorage` class consists of a field, a constructor, three data management methods, and a `main` method. Each component is described below, including its purpose, technical details, and role in the class's functionality.

1. **Field:**

  - **`private Map<Integer, Patient> patientMap`**

    - **Purpose:** Stores `Patient` objects indexed by their unique `patientId`, serving as the primary data structure for organizing patient data.

    - **Technical Details:**

      - Type: `Map<Integer, Patient>`, implemented as a `HashMap` for O(1) lookup, insertion, and deletion by `patientId`.

      - Private access ensures encapsulation, initialized in the constructor and modified via `addPatientData` and accessed via `getRecords` and `getAllPatients`.

      - Keys are `Integer` (boxed `int` patient IDs), and values are `Patient` objects containing patient-specific records.

    - **Role:** Provides efficient storage and retrieval of patient data, enabling the system to manage multiple patients and their records.

2. **Constructor:**

  - **`public DataStorage()`**

    - **Purpose:** Initializes a new `DataStorage` instance with an empty `HashMap` for storing patient data.

  - **Technical Details:**

   - No parameters, as the class starts with an empty state.

   - Logic: Initializes `patientMap` as a new `HashMap<Integer, Patient>`.

   - Public access allows instantiation by `HealthDataSimulator` or other components (e.g., the `main` method).

  - **Role:** Prepares the `DataStorage` instance for use, setting up the underlying storage structure to accept patient data.

3. **Methods:**

 - **`public void addPatientData(int patientId, double measurementValue, String recordType, long timestamp)`**

  - **Purpose:** Adds or updates patient data by creating or retrieving a `Patient` object and adding a new record to its record list.

  - **Technical Details:**

   - Parameters:

    - `patientId` (`int`), the unique identifier of the patient.

    - `measurementValue` (`double`), the numerical value of the health metric (e.g., 120.0 for blood pressure).

    - `recordType` (`String`), the type of measurement (e.g., "HeartRate", "BloodPressure").

    - `timestamp` (`long`), the time of measurement in milliseconds since the Unix epoch.

   - Return Type: `void`, as the method’s effect is updating `patientMap` and the patient’s records.

   - Logic:

    - Retrieves the `Patient` for `patientId` from `patientMap` using `get`.

    - If no `Patient` exists (`null`), creates a new `Patient` with `patientId` and adds it to `patientMap` using `put`.

    - Calls `patient.addRecord` to add a new `PatientRecord` with the provided `measurementValue`, `recordType`, and `timestamp`.

- Public access allows data generators or data readers (e.g., `DataReader`) to store data via `DataStorage`.

- Does not validate inputs (e.g., positive `patientId`, non-null `recordType`), assuming valid data from callers.

- **Role:** Enables the storage of health data, creating or updating patient records as data is generated or ingested, forming the basis for data management.

- **`public List<PatientRecord> getRecords(int patientId, long startTime, long endTime)`**

  - **Purpose:** Retrieves a list of `PatientRecord` objects for a specific patient, filtered by a time range.

  - **Technical Details:**

    - Parameters:

      - `patientId` (`int`), the unique identifier of the patient.

      - `startTime` (`long`), the start of the time range in milliseconds since the Unix epoch.

      - `endTime` (`long`), the end of the time range in milliseconds since the Unix epoch.

    - Return Type: `List<PatientRecord>`, a list of records within `[startTime, endTime]` or an empty list if the patient is not found.

    - Logic:

      - Retrieves the `Patient` for `patientId` from `patientMap` using `get`.

      - If the `Patient` exists, calls `patient.getRecords(startTime, endTime)` to get filtered records.

      - If no `Patient` is found, returns an empty `ArrayList`.

    - Public access allows `com.alerts.AlertGenerator`, monitoring tools, or other components to retrieve records for analysis or alert generation.

    - Does not validate `patientId`, `startTime`, or `endTime` (e.g., ensuring `startTime <= endTime`), relying on `Patient.getRecords` for filtering logic.

  - **Role:** Facilitates time-based data retrieval for a specific patient, supporting analysis of health trends or alert generation based on historical data.

- **`public List<Patient> getAllPatients()`**

  - **Purpose:** Retrieves a list of all `Patient` objects stored in the `DataStorage`.

  - **Technical Details:**

    - No parameters, as it returns all patients.

    - Return Type: `List<Patient>`, a new `ArrayList` containing all `Patient` objects from `patientMap.values()`.

    - Logic: Creates a new `ArrayList` from `patientMap.values()` to avoid exposing the internal `HashMap`'s collection.

    - Public access allows `com.alerts.AlertGenerator` or monitoring tools to iterate over all patients for system-wide analysis or alert evaluation.

  - **Role:** Enables access to all patient data, supporting bulk operations like evaluating all patients for alerts or generating summary statistics.

- **`public static void main(String[] args)`**

  - **Purpose:** Demonstrates the usage of `DataStorage` by initializing the system, simulating data ingestion, retrieving records, and triggering alert evaluations.

  - **Technical Details:**

    - Parameters: `args` (`String[]`), command-line arguments (unused in the provided code).

    - Return Type: `void`, as it is a demonstration method.

    - Logic:

      - Comments indicate an undefined `DataReader` that should read data into `storage` (e.g., from a file or network source), but it is not implemented.

      - Creates a new `DataStorage` instance.

      - Retrieves records for `patientId` 1 within a hardcoded time range (1700000000000L to 1800000000000L) and prints them to `System.out`, showing `patientId`, `recordType`, `measurementValue`, and `timestamp`.

      - Creates an `AlertGenerator` instance with the `DataStorage`.

      - Iterates over all patients (via `getAllPatients`) and calls `alertGenerator.evaluateData` to check for alert conditions.

- Static and public access allows execution as a standalone program for testing or demonstration.

- References an undefined `DataReader` and `PatientRecord` methods (e.g., `getPatientId`, `getRecordType`), indicating dependencies on other classes.

- **Role:** Provides a proof-of-concept for `DataStorage` usage, demonstrating data storage, retrieval, and integration with the alert system, though it requires additional components (`DataReader`) to be fully functional.

---

#### Technical Points and Design Considerations

1. **Data Organization and Efficiency:**

- The use of a `HashMap<Integer, Patient>` ensures O(1) access to `Patient` objects by `patientId`, making `addPatientData` and `getRecords` efficient for patient lookup. The `ArrayList` in `Patient` for records complements this by providing O(1) appends and O(n) retrieval, suitable for typical simulation workloads.

- The separation of concerns between `DataStorage` (managing patients) and `Patient` (managing records) promotes modularity, though the redundant `patientId` in `PatientRecord` could be optimized by relying on the `Patient` context.

2. **Thread Safety:**

- The `patientMap` (`HashMap`) is not thread-safe, and concurrent calls to `addPatientData`, `getRecords`, or `getAllPatients` in `HealthDataSimulator`'s multi-threaded environment could cause race conditions (e.g., `ConcurrentModificationException` or inconsistent `Patient` creation).

- To ensure thread safety, `patientMap` could be replaced with a `ConcurrentHashMap`, or methods could be synchronized, though this may introduce performance overhead. Alternatively, `DataStorage` could be designed as a singleton or use a dedicated writer thread to serialize updates.

3. **Error Handling and Validation:**

- The class lacks input validation for `patientId`, `measurementValue`, `recordType`, `timestamp` (`addPatientData`), and `startTime`/`endTime` (`getRecords`). For example, negative `patientId`, null `recordType`, or invalid time ranges could lead to inconsistent data. Adding validation (e.g., throwing `IllegalArgumentException`) would improve robustness.

- The `getRecords` method gracefully handles missing patients by returning an empty list, but it could log such cases for debugging. The `main` method assumes `PatientRecord` methods exist, indicating incomplete integration.

4. **Performance Considerations:**

- The `addPatientData` method is efficient (O(1) for `HashMap` operations, O(1) for `Patient.addRecord`), but `getRecords` depends on `Patient.getRecords`' O(n) iteration, which could be slow for large record sets. Indexing records by timestamp (e.g., in a `TreeMap`) or using a database could improve retrieval performance.

- The `getAllPatients` method creates a new `ArrayList`, which is O(n) for n patients but avoids exposing `patientMap`'s internal collection, ensuring encapsulation.

- For large-scale simulations, integrating with a database (e.g., SQLite, PostgreSQL) via JDBC could enhance scalability and persistence.

5. **Integration with Task 7 Design Patterns:**

- The `DataStorage` class is central to Task 7's alert system, providing data to `com.alerts.AlertGenerator` via `getRecords` and `getAllPatients`. The `evaluateData` method in `AlertGenerator` processes `Patient` records to trigger alerts, which are created by `AlertFactory` subclasses (e.g., `ECGAlertFactory`) and enhanced by `AlertDecorator` subclasses (e.g., `PriorityAlertDecorator`).

- The generic `recordType` in `addPatientData` supports diverse data types, including alerts (e.g., `recordType` "Alert"), aligning with Task 7's flexible alert processing.

- The class integrates with the Strategy Pattern via `OutputStrategy` implementations (e.g., `WebSocketOutputStrategy`), which may output records retrieved by `getRecords` for real-time monitoring or logging, supporting alert dissemination.

6. **Potential Improvements:**

   - Add input validation for `patientId` (positive), `recordType` (non-null), `timestamp` (non-negative), and time ranges (`startTime <= endTime`) to throw `IllegalArgumentException` for invalid values.

   - Make `patientMap` thread-safe (e.g., using `ConcurrentHashMap`) to support concurrent access in multi-threaded environments.

   - Optimize `getRecords` for large datasets by indexing records in `Patient` or integrating with a database for faster queries.

   - Implement a `removePatient` or `pruneRecords` method to manage memory by deleting old data or unused patients in long-running simulations.

   - Replace `System.out` in the `main` method with a logging framework (e.g., SLF4J) for structured logging and better debugging.

   - Define an interface for `DataStorage` (e.g., `DataStore`) to support alternative implementations (e.g., database-backed storage) via dependency injection.

   - Complete the `main` method by implementing or mocking `DataReader` and ensuring `PatientRecord` method compatibility (e.g., `getPatientId`, `getRecordType`).

---

#### Interaction with Other Components

- **With `Patient` Class:** The `DataStorage` class manages a collection of `Patient` objects, creating them in `addPatientData` and delegating record storage (`addRecord`) and retrieval (`getRecords`) to `Patient` instances. The `getAllPatients` method returns `Patient` objects for system-wide operations.

- **With `PatientRecord` Class:** `DataStorage` indirectly interacts with `PatientRecord` via `Patient`, as `addPatientData` creates records through `Patient.addRecord`, and `getRecords` retrieves `PatientRecord` objects. The `main` method assumes `PatientRecord` methods like `getPatientId` and `getRecordType`.

- **With `PatientDataGenerator` Implementations:** Data generators (e.g., `ECGDataGenerator`, `BloodPressureDataGenerator`) in `com.cardio_generator.generators` produce data that `HealthDataSimulator` stores in `DataStorage` via `addPatientData`, mapping generator output to `measurementValue`, `recordType`, and `timestamp`.

- **With `com.alerts.AlertGenerator` (Task 7):** The `DataStorage` class provides data to `AlertGenerator` via `getRecords` (for specific patients) and `getAllPatients` (for system-wide evaluation). `AlertGenerator` processes records to trigger alerts, which are created by `AlertFactory` subclasses and enhanced by `AlertDecorator` subclasses.

- **With `Alert` Class (Task 7):** Alerts generated from `DataStorage` records (e.g., stored as `recordType` "Alert") are represented as `Alert` objects (from `com.alerts`), supporting Task 7's Factory and Decorator patterns.

- **With `OutputStrategy` Implementations:** Records retrieved by `getRecords` can be output via `OutputStrategy` implementations (e.g., `WebSocketOutputStrategy`, `FileOutputStrategy`) for real-time monitoring or logging, supporting Task 7's alert dissemination.

- **With `DataReader` (Undefined):** The `main` method references an undefined `DataReader` class, likely responsible for reading external data (e.g., from files or networks) into `DataStorage` via `addPatientData`. This indicates integration with external data sources, though it requires implementation.

---

#### Summary of Functionality

The `DataStorage` class is the central hub of the cardiovascular data simulator's data management subsystem, organizing patient data in a `HashMap` of `Patient` objects and providing methods to add data (`addPatientData`), retrieve records (`getRecords`), and access all patients (`getAllPatients`). It integrates with `HealthDataSimulator` to store data from `PatientDataGenerator` implementations and supports Task 7's alert system by supplying data to `com.alerts.AlertGenerator` for evaluation, compatible with `AlertFactory` and `AlertDecorator`. The class's use of `HashMap` ensures efficient patient access, and its `main` method demonstrates usage, though it requires a `DataReader` implementation. The class is effective for managing clinical data, but enhancements in thread safety, validation, and performance optimization could improve its scalability for large-scale simulations. Overall, `DataStorage` plays a pivotal role in enabling structured data management, supporting real-time monitoring, historical analysis, and alert generation in the simulator.