1. Abstract class/ inheritence usage in my project

In my term project, I utilized abstract classes and interfaces to create a modular and extensible design for managing health data. I defined an abstract class called HealthData that served as a base class for different types of health data, such as CommonHealthData and CustomHealthData. This allowed me to define common properties and methods that are shared by all health data types, while also providing flexibility to add new data types in the future i.e exercise data, sleepingdata, heartratedata, temperaturedata, etc.

   a. Used abstract class Healthdata with getdata and getmetric as abstract method. Why didn't I use interface? Remember Interafce will only tell what needs to be implemented. Abstract class also kind of does the same but I can implement some functions in it. The commonhealth and customhealth data class extends healthdata class and implements the abstract methods in it. It can also use name date attributes from the healthdata class.
   b. Abstract classes allowed me to define common implementation details for methods and attributes. By using an abstract class, I can provide default behavior or shared functionality among the different health data types. For example, the HealthData abstract class can define methods for data validation or common data manipulation operations that are applicable to all subclasses.

   Abstract classes and interfaces have likely played a significant role in structuring and organizing my project. They provide a foundation for defining common behaviors, contracts, and relationships between different components. By promoting code reuse, encapsulation, and polymorphism, they facilitate a more scalable, usable, and maintainable application architecture. Reflect on specific instances in my project where abstract classes and interfaces were instrumental in achieving these objectives and highlight their impact on the overall success of my project.

2. Used validate() custom healthdataexception as user defined. I defined the class that extends Exception and the constructor takes whatever String is passed to the the healthdataexception() when it gets thrown.
   a. Ensure that each custom exception class provides clear and meaningful error messages, helping developers or users understand the cause of the exception.
   b. Provide informative error messages and stack traces in my exceptions to aid in debugging and troubleshooting.
   c. Create exception hierarchies by introducing additional abstract or concrete exception classes to handle different levels of exceptions, such as HealthDataRuntimeException and HealthDataCheckedException.

Exception handling played a crucial role in my project, particularly user-defined exceptions. For example, I created a custom exception called HealthDataException to handle specific errors related to health data processing. This allowed me to catch and handle exceptions in a more granular and meaningful way, providing informative error messages to the user. Exception handling helped improve the robustness and user-friendliness of my application by gracefully handling unexpected situations and providing appropriate feedback to the user.

3. Usage of generics in the healthdata<> is to show it can be type independent but I think the polymorphism was the better option here. But, in the main class where I had to use a lot of casting, I think it would have been helpful there. The use of the T type for storing health data in the list.
    a. Use generics when the classes, methods are repetitive with slight changes.
    b. Use a class that takes <T extends class x> then create objects out of it to access the methods defined in the classes that are not common, if the methods were common, as defined in the super class, I could use poly but would need casting.
    c. Type safety: Generics help enforce type constraints at compile-time, reducing the likelihood of type-related errors and providing better code correctness.
    d. Code reusability: With generics, I can write generic algorithms or methods that can work with multiple types of health data, promoting code reuse across different scenarios.
    e. Flexibility: Generics allow I to write more flexible and adaptable code, as I can instantiate classes or collections with different types of health data dynamically.

    Generics can be valuable when enforcing type constraints or performing input validation. For instance, when I accept user input for health data, I could have used generics to define generic input validation methods that can be applied to different types of health data inputs.

    By parameterizing the HealthData class with a wildcard (<?>), the compiler ensures that only valid HealthData objects or their subclasses can be stored in the list. This helps prevent type-related errors at compile-time and provides type safety when accessing the properties and methods of each health data object within the loop.

    I made use of generics in several classes and methods within my project. For instance, the HealthData class was defined with a generic type parameter T to support different types of health data. This allowed for type safety and ensured that the correct data type is used when working with health data objects. Additionally, I used generics in methods that processed and retrieved health data, ensuring type compatibility and enabling flexibility in handling different data types. I think the best use of it was when I used it in the arraylist collection.

4. Streams and lambdas were not extensively used in my project due to the use of JavaFX for user interface development. However, I could have leveraged streams and lambdas more effectively for data manipulation and filtering if my project involved bulk data processing or complex data transformations.
    a. Enhance the user experience by adding more interactive features, such as tooltips, animations, and drag-and-drop functionality, to make the application more engaging.
    b. Implement data validation and error handling to provide informative feedback to the user when entering health data.
    c. Utilize JavaFX's threading capabilities to perform time-consuming operations, such as database queries or data processing, in background threads to keep the UI responsive.
    d. Improve the laIt and organization of the user interface to ensure a logical flow and easy navigation for users.

e. Refactor my code to follow the Model-View-Controller (MVC) or a similar architectural pattern to separate concerns and make the application more maintainable and scalable.
f. Consider modularizing my codebase by breaking it into reusable components or modules, promoting code reuse and easier management of larger applications.

5. I used the javafx related platform.runlater feature for opening and closing gui windows. Concurrency was not a prominent feature in my project as it primarily focused on health data management for a single user. However, if scalability and multi-user support were requirements, I could have incorporated concurrency features to handle concurrent data access and processing. For example, I could have used threads or thread pools to perform background tasks without blocking the user interface, ensuring smooth user experience even with multiple concurrent operations.

    a. Leveraging Threads:
    b. I could have used separate threads to perform time-consuming tasks such as database operations, file I/O, or complex computations, keeping the UI responsive. This would prevent blocking the UI thread and enhance the overall user experience.
    c. For example, when retrieving health data from a database, I could have used a background thread to execute the database query and populate the UI once the data is fetched.
    d. By using threads, I can take advantage of multi-core processors and improve the performance and efficiency of my application.
    e. Thread Pools:
    f. Instead of creating threads manually for each task, I could have employed thread pools to manage and reuse threads efficiently.
    g. Thread pools provide a predefined set of worker threads that can be assigned to tasks as needed, reducing the overhead of creating and destroying threads.
    h. This approach is particularly useful when I have a large number of short-lived tasks to execute concurrently.
    i. For example, if my application allows multiple health data entries at the same time, I could have used a thread pool to handle each entry operation.
    j. Maintaining and Scaling Concurrency:
    k. When using concurrency, it is essential to handle synchronization and thread safety to prevent data corruption or race conditions.
    l. Ensure proper synchronization mechanisms, such as locks or synchronized blocks, when accessing shared resources to maintain data integrity.
    m. Use thread-safe data structures or apply proper synchronization techniques to handle concurrent access to data collections or shared variables.
    n. Scalability can be achieved by fine-tuning the thread pool configuration based on the available hardware resources and the nature of the tasks.
    o. I can also consider using higher-level concurrency constructs, such as CompletableFuture or the java.util.concurrent package, to handle more complex scenarios.
    p. Choosing the Right Concurrency Approach:

q. Evaluate the specific requirements and characteristics of my application to determine the most suitable concurrency approach.
r. Consider factors such as the type and frequency of tasks, the level of parallelism required, and the potential impact on the user experience.
s. For example, if my application involves heavy computation or long-running tasks, utilizing parallel streams or asynchronous programming with CompletableFuture could lead to performance improvements.
t. Data Processing and Validation:
u. When the user submits health data, I could have used concurrency to offload the processing and validation tasks to separate threads.
v. By doing so, I could have ensured that the application remains responsive during the processing, especially if the validation involves computationally intensive operations or external dependencies.
w. Concurrency would have allowed the application to continue accepting user input or performing other tasks while concurrently processing and validating the health data.

It's worth noting that the use of concurrency should be carefully considered based on the specific requirements and limitations of my application. Since my application has a relatively small scope and user base, and the operations described are not computationally intensive, the need for extensive concurrency may be limited. However, these are some areas where concurrency could have been applied to enhance performance and user experience within the given functional scope of my application.