OOP Mini Project

Specification

In this mini-project, your are asked to write a small library implementing a basic backend for an online marketplace. Users of the marketplace—identified by a unique username—can be buyers or sellers: sellers list items for sale and buyers bid for them. Below is the typical lifecycle of a listing:

1. The seller creates the listing (in *draft* state). At creation, the listing is assigned a unique ID by the library. The following data needs to be specified for a listing: title (limited length), starting price, description (unlimited length), minimum bidding time. Some or all of this data can be specified and modified after creation.
2. The seller can change the listing state from draft to *active* at any time, as long as all data from point 1 has been specified. Alternatively, the seller can change the lifting state from draft to *cancelled*.
3. When the listing is active, buyers can submit bids. Only the current highest bid is listed. Each buyer can have at most one bid on the listing at any time and can withdraw their bid at any time. A new bid can be submitted only if it is higher than the highest current bid.
4. After the minimum bidding time has elapsed from the moment the listing has become active, the seller can change the listing state from active to *sold*, as long as at least one bid is present: when this happens, the buyer with the highest bid has bought the item. At any time when no bids are present, the seller can change the listing state from active to cancelled.

The library should offer the following minimum functionality:

- Creation of listings, changes to listing state, submission and management of bids on the listing. Quick access to listing by ID.
- Search for active listings. It should be optionally possible to filter listings by some criteria of your choice (e.g. title containing given string, highest bid in given range, remaining bidding time above a given minimum, etc). It should be optionally possible to sort listings by some criteria of your choice (e.g. title, highest bid, remaining bidding time), both in ascending order and in descending order.
- For a given seller: access their draft, active, sold and cancelled listings. Also, track the total amount of money they made from sold listings so far.
- For a given buyer: access their current bids on active listings and the listings they have bought. Also, track the total amount of money they spent on bought items so far.

To manage time, use the built-in [Date]() class. A class `Marketplace` should act as the entry point to your library (i.e. as a [façade]()). If you feel like a challenge, you can implement one or more of the following stretch goals:

- Create a generic data structure to use for keeping listing bids, only allowing the legal bid management operations.
- Introduce a subscription functionality, where other processes can request to be notified of these events:
  - a new listing has become active
  - a given listing has a new highest bid
  - a given listing was sold or cancelled

Using the second and third events above, you might also want to try implementing a budget functionality that for a given buyer tracks the amount of money currently on highest bids for active listings.

- Introduce an undo function in the editing of draft listings. Allow new listings to be created using old ones as a starting point (a blueprint, if you will).
- Introduce advanced search functionality on listings, allowing users of the library to specify their own criteria/logic (e.g. by supplying a filter function).
- Introduce prices with specified currency and currency conversion functionality (say GBP, EUR, USD and JPY). You can base conversion on a global forex object, with a method providing live exchange rates for a given currency pair: in reality, this would use a connection to some online service, but you should just return some fixed numbers :)

Task

Design and implement a library with the functionality described above, using an object oriented approach:

- Low-level data used by the library should be structured by means of suitable types (try to avoid classes for lightweight data) and validated where necessary. As much data validation as possible should be delegated to the static type-checking. Avoid overly broad low-level types (e.g. use `bigint` instead of `number` for integers, use unions of literal types for finite collection of values, etc).
- High-level components used by the library should be structured through interfaces and classes, with public methods providing access to external functionality and private/protected methods providing access to internal functionality.
- Your design should make use of adequately chosen types, interfaces and access control to ensure that the library cannot be misused, while allowing individual components to be safely exposed to the users.
- Where relevant, you should use advanced type features (e.g. polymorphism, advanced types, type operators) and idiomatic language features (e.g. destructuring, `for..of` loops, `for..in` loops, functional methods on arrays).
- Where relevant, you should implement reusable generic data structures (e.g. trees, graphs, queues/stacks).

Your code should be clear and concise: long or complex method/function bodies should be avoided whenever possible (e.g. by delegating to helper methods/functions, or to other components). Where long or complex code is unavoidable, the individual steps should be concisely documented using single-line comments.