Section 1

For the time section of the assignment I created a `Time` class with some essential properties and helper methods. With the essential properties of `origin`, `ticSize`, `anchor`, and `age`. The "current time" is stored in `tic`. The method to get calculate `tic` is in `getTime()` created. These things are stored in a class because the time should be thought of as a service other objects can query

The time is calculated by taking the difference of the current time from the value of `lastTime` and dividing by the tic size. I've also included a few helper methods. `pause()` increases the tic size to the maximum int value, `resume()` returns it to the original value, and `advanced()` is used in my other implementations to check if time had advanced since the last time it was called (this is used so redundant updating doesn't happen when time hasn't changed since the last loop iteration).

In the game I created three versions of time: real time (uses `System.nanoTime`), game time, and replay time. Real time is anchored to the machine clock, game time is anchored to real time with a tic size that increments in millseconds, and replay time which is anchored to real time and initially has the same tic size.

In previous versions of the homework I moved objects based on how many times update was called. But in this assignment things are moving based on the time difference from when an update was called. This "last time" and "difference" logic was moved into the game object model into `Movable`. And when update methods use `continueUpdate()` the time difference between calls is calculated and is used by subclasses to move by their `movementFactor` (this controls the magnitude of movement, the lower the number the more magnitude).

Section 2

To facilitate events I created an `EventManager` class which stores an event priority queue, a hashmap for registration, and three methods for registration, raising, and handling.

Registration: I used a hashmap where the keys are event types and the values are an arraylist of `GameObjec`. My event types were strings (for flexibility) and this is the naming scheme: "keyboard<id>", "collision<id>", "spawn<id>", "death<id>", and "*". "<id>" is the id of Client. I'm adding the id to the type because in handling all keyboard events from all client would get sent to every client. Doing it this way makes sure a particular client's events are handled by that particular client's objects. "*" is a wildcard so the replay system can get registered for all the events. Strings afford more specific event types in contrast to doing something with event type inheritance where it's more difficult to change event types because the structure is more rigid.

All handlers are registered in `Server` when it starts and at each time a new `Client` is started. Using an arraylist as the hashmap values were a good choice because the number of handlers for each event type is unknown and so more flexible dynamic arrays would fit this.

Raising: Whatever object that created the conditions necessary for an event will create an `Event` and use the `EventManager` to add it to the priority queue. On event creation the invoking object will pass in an event type, object parameters (of data type `Object` for flexibility), age, and priority. Then when the event reaches the event manager raising method, which is on the server side, the timestamp is filled in. This is done on the server because clients don't have access to gametime because they don't have access to the correct realtime. At this point the events are compared based on timestamp and priority and then added to the priority event queue. `Event` implements the `Comparable` interface for comparison. When an event generates a new event: for example collision creates a death, then the death event is raised in the handler that handled the collision in the exact same execution stack.

Handling: in the last homework `GameObject` was just a marker interface. However, in this assignment I was able to find something more useful for it (as I guessed). I added the `.onEvent(Event e)` to `GameObject` and was a natural fit into my game object model. This type of handling compared to using statically typed methods is more suitable to a component based game object model, like the one I have. Statically typed methods are better for hierarchy based models.

In the main game loop the event manager invokes `handle()` and then executes all the events in the queue by invoking each handler's .onEvent(). This is not a good design because handling all the possible events in one game loop could really delay things if there are a lot of events. You could solve this by alotting a fixed time to the event manager so to interrupt handling and allow the rest of the game loop to continue. However, for the small scope of this game that would be premature optimization.

There's one significant design change I made that doesn't specifically involve the event system. Previously I had one `Character` but then I found there were some Server side components a Character should know about but the client couldn't get to. So I made two versions of the character: `CharacterServer` and `CharacterClient`. The main difference is the server character implements `.onEvent` and has a reference to the event manager. Since a character constructor needs to be called on the client side it would have no way to reference an event manager and so two classes needed to be created. I think this is a reasonable design because the server and client division is a natural one to think about as there are physical limitations. In network applications I'm sure server and client versions of things are common and necessary.

Section 3

I started to try and build this replay system but ran out of time and debugging motivation. I created the `Replay` class which holds the event log and some properties and methods for managing the replay state.

The replay is started immediately after recording is stopped and then normal play resumes when it's done. A new replay time is created when the replay starts and its origin is set to sometime before the timestamp of the first event. Then the part of the game loop that updates objects and runs the event handler is switched into replay mode and runs through the events of age less than 1 on the log. It's at this point where I ran into problems.

I was never able to get the timing right. Events were processed too fast or too slow so it never ran for as long as the recording was. I'm fairly confident the way I'm timestamping things is fine so I think there's more of a problem in how and when the events are handled. As of the time of submission the replay will execute some of the events but not all. There's some replay movement (even from the floating platform) but not all the events are executed.

I do want to mention about how I thought about how I would have done a few things if I didn't get stuck. I think to handle the events that were created by other events (ages greater than 0) you could create another queue just for these type of events and use it to execute all the events when it's not empty and then proceed with the normal log when done. Also running the replay at different speeds would have been easy to do with my design all you would need to do is read client input and change the replay time tic size. The time changes would then automatically take effect in the event handles.

Now that you have designed and implemented a number of engine components, write a 2–3 page paper **summarizing your design**. That is a minimum of 2 FULL single-spaced pages. Think creatively about what you have done. **Why did you make the design decisions you made? What did they enable for this assignment? What will they enable in the future? What didn't work out the way you had hoped, and why**? The most successful writeups will contain evidence that you have thought deeply about these decisions and implementations and what they can produce and have **gone beyond what is written in the assignment**. As an appendix to your paper, please include all relevant screenshots to support your discussion. The appendix does not count toward your 2–3 page requirement. Your points for the writeup will represent 1/2 of the points allocation for each of the above sections