UMassAmherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 11: Observables

Wednesday November 12, 2025

Reminders

- HW 6 is released! You should have received an email last night from CATME with information about your assigned team
    - Reach out early and begin coordinating time to work together!
- Midterm 2 grades have been released
    - Regrade requests will be open until Wednesday 11/19
    - Request a regrade if you feel unsure about the grading or believe there was a mistake made in grading

Today’s Goals

- Homework 6 setup
- Observables
- Observers

Homework 6: Do these now!

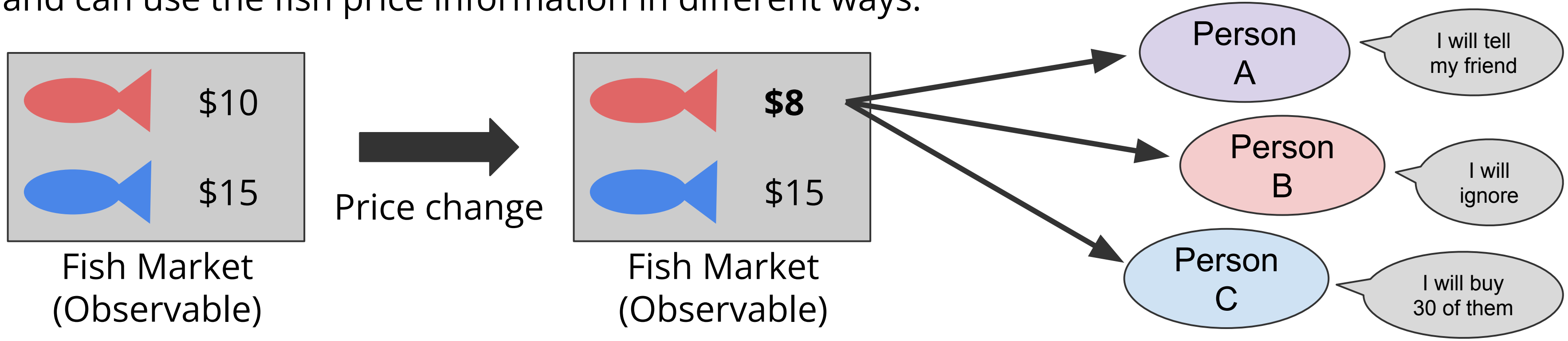
1. **PLEASE** disable the jest extension in VSCode if you are using it. An alternative extension would be the jest-runner extension. See the HW specs if you are curious why!
2. If you do not have a Github account under your UMass email:
   - Check your email for an invite to a Github repository. When you click it, you can login with an existing GitHub account, or you can create a new one
   - It is not required that your GitHub account uses your UMass email, personal accounts are acceptable
3. If you already have a Github account under your UMass email, you may need to check your notifications in Github for the invite
4. Please accept the Github invite before it expires!

Observables Review

The Observer design pattern describes two main components, Observers and Observables.

Observables are like publishers, which broadcast information to their "audience". The members of this audience are Observers. Observers subscribe to Observables in order to receive the broadcasted information.

For example, I could have an Observable for the price of different fish in a market. Each time the price of a fish changes, I will broadcast the new price to anyone who is interested. Specifically, people (Observers) can indicate their interest by subscribing to my Observable, and can use the fish price information in different ways.



UMassAmherst | Manning College of Information & Computer Sciences

Observables Review

```
interface Observable {
  // adds the given subscriber to the set of subscribers
  subscribe(subscriber: Observer): void;

  // removes the given subscriber from the set of subscribers
  unsubscribe(subscriber: Observer): void;
}
```

The Observer design pattern is defined differently in many contexts, but in this class we will assume Observables have the above interface. Observers do not have a set interface, and will be defined based on the context.

We will typically assume both Observables and Observers to be objects, but there are instances where it can make more sense for Observers to be functions! See the lecture notes for more details.

Observables Review

```
interface Observable {
  // adds the given subscriber to the set of subscribers
  subscribe(subscriber: Observer): void;

  // removes the given subscriber from the set of subscribers
  unsubscribe(subscriber: Observer): void;
}
```

Note: if you have started working on the extra credit homework, you will have noticed that Observables and Observers are defined differently in that assignment compared to lecture.

These definitions are not too distinct, but to avoid confusion, any lecture, lab, or exam questions on Observables will use the above interface and treat Observers as objects, unless otherwise stated.

The extra credit homework gives you a chance to explore a slightly different definition of Observables, and how you can essentially compose Observables in different scenarios.

UMassAmherst | Manning College of Information & Computer Sciences

Context of Today’s Exercises

You will be implementing an Observable and two Observers for handling different (computer) mouse events. Specifically, mouse clicks and mouse drags. Mouse clicks are described by the position (coordinate) clicked, and mouse drags are described by the start and end positions when the mouse is dragged.

```
interface Observable {
  // adds an Observer to the set of subscribers
  subscribe(subscriber: MouseObserver): void;

  // removes an Observer from the set of subscribers
  unsubscribe(subscriber: MouseObserver): void;
}
```

Context of Today's Exercises

The MouseObserver interface has a readonly property **screen** that describes which screen this observer belongs to. Mouse events (both clicking and dragging) take place on a specific screen, and a MouseObserver should only receives updates for their corresponding screen.

Think of having 2 monitors connected to your laptop; in total you would have 3 screens where your mouse can click or drag.

```
interface MouseObserver {
  readonly screen: number;

  // update observer with new mouse click coordinate
  update_coordinate(coord: Coordinate): void;

  // update observer with new mouse drag event
  update_drag(start: Coordinate, end: Coordinate): void;
}
```

Assume screen numbers start at 0 and count upwards. I.e. if there are 3 screens total, the valid screen numbers would be 0, 1, 2.

Coordinate is defined as {x: number, y: number}. You can also assume that both x and y are non-negative.

Exercise 1: Observables

- Create a class **MouseEvents** that is an Observable, implementing the following:
  - **`constructor`**`(num_screens: number)`
    - Initializes number of screens of MouseEvents and any other attributes
  - **`subscribe`**`(subscriber: MouseObserver): void`
    - Adds the given observer to a set of observers
  - **`unsubscribe`**`(subscriber: MouseObserver): void`
    - Removes the given observer if they are subscribed
  - **`mouse_click`**`(screen: number, coord: Coordinate): void`
    - Updates all observers for the given screen with the mouse click Coordinate
  - **`mouse_drag`**`(screen: number, start: Coordinate, end: Coordinate): void`
    - Updates all observers for the given screen with the start and end coordinates of the mouse drag
- Use the definitions at the top of the starter code! Try to reduce code duplication and think about how to broadcast updates efficiently.

Exercise 1: Solution

```
class MouseEvents implements Observable {
  private _num_screens: number;
  private _subscribers: Set<MouseObserver>[];

  constructor(num_screens: number) {
   this._num_screens = num_screens;
   this._subscribers = [];
   // ...
  }

  // ...
}
```

We declare two private attributes, one to store the number of screens and one to store our subscribers. Since we want to updates subscribers based on which screen they are subscribed to, we will store subscribers in different sets depending on their screen.

Exercise 1: Solution

```
class MouseEvents implements Observable {
  private _num_screens: number;
  private _subscribers: Set<MouseObserver>[];

  constructor(num_screens: number) {
   this._num_screens = num_screens;
   this._subscribers = [];
   for (let i = 0; i < num_screens; i++) {
    const new_set = new Set<MouseObserver>();
    this._subscribers.push(new_set);
   }
  }

  // ...
}
```

We initialize this array of sets by creating a new (empty) set for each screen number.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 1: Solution

```
class MouseEvents implements Observable {
  private _num_screens: number;
  private _subscribers: Set<MouseObserver>[];

  private _is_valid_screen(screen: number): boolean {
    return screen >= 0 && screen < this._num_screens;
  }

  subscribe(subscriber: MouseObserver): void {
    if (this._is_valid_screen(subscriber.screen)) {
      this._subscribers[subscriber.screen].add(subscriber);
    }
  }
  // ...
}
```

We will define a private helper method to check if a given screen number is valid for this Observable. We then define the subscribe method, adding the given subscriber to the corresponding set if their screen number is valid.

Exercise 1: Solution

UMassAmherst | Manning College of Information & Computer Sciences

```
class MouseEvents implements Observable {
  private _num_screens: number;
  private _subscribers: Set<MouseObserver>[];

  unsubscribe(subscriber: MouseObserver): void {
    if (this._is_valid_screen(subscriber.screen)) {
      this._subscribers[subscriber.screen].delete(subscriber);
    }
  }

  // ...
}
```

We follow a very similar approach for unsubscribe. Note that for Sets in JS/TS, .add() adds an element to a set, and does nothing if it already exists in the set. Meanwhile, .delete() removes an element from a set, and does nothing if it is not in the set.

This would be a great place to reduce code duplication! See the solutions file for more.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 1: Solution

```
class MouseEvents implements Observable {
  private _num_screens: number;
  private _subscribers: Set<MouseObserver>[];

  mouse_click(screen: number, coord: Coordinate): void {
    if (this._is_valid_screen(screen)) {
      for (const subscriber of this._subscribers[screen]) {
        subscriber.update_coordinate(coord);
      }
    }
  }
  // ...
}
```

For mouse_click, we must first check if the given screen is valid. If it is, then **this._subscribers[screen]** will give us a set of all subscribers who we should update. We then go through each of these and call the corresponding update method, passing in the coordinate the mouse was clicked at.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 1: Solution

```
class MouseEvents implements Observable {
  private _num_screens: number;
  private _subscribers: Set<MouseObserver>[];

  mouse_drag(screen: number, start: Coordinate, end: Coordinate): void {
    if (this._is_valid_screen(screen)) {
      for (const subscriber of this._subscribers[screen]) {
        subscriber.update_drag(start, end);
      }
    }
  }
}
```

We follow a very similar approach for mouse_drag. Again, note that this is a great place to reduce code duplication! See the solutions file for more (posted after today).

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 2a: Observers

- Implement the MouseEventLogger Observer, with the following methods:
  - **`update_coordinate`**`(coord: Coordinate): void`
    - Takes the given coordinate and prints out the string “`Click: (x, y)`”, with x and y replaced by the respective coordinates in `coord`.
  - **`update_drag`**`(start: Coordinate, end: Coordinate): void`
    - Takes the given coordinates and prints out the string “`Drag: (x1, y1) to (x2, y2)`”, with x1 and y1 replaced by the start coordinates, and x2 and y2 replaced by the end coordinates.
- The constructor has already been implemented for you
- Your implementations of these methods should pretty much just be one line each!

Exercise 2b: Observers

- Implement the MouseEventArea Observer, with the following methods:
  - **`constructor`**`(screen: number)`
    - Initializes the screen number of this MouseObserver, and any other attributes
  - **`update_coordinate`**`(coord: Coordinate): void`
    - Calculates the area of the smallest (not rotated) rectangle that contains all coordinates clicked on so far
    - Prints "`Coordinate area: A`", with A replaced by the calculated area.
    - Hint: consider the rectangle formed by (x_min, y_min) and (x_max, y_max) as opposite corners, where x_min, y_min correspond to the smallest x or y values seen so far
  - **`update_drag`**`(start: Coordinate, end: Coordinate): void`
    - Calculates the area of the rectangle formed by the start and end coordinates as opposite corners.
    - Prints "`Dragged area: A`", with A replaced by the calculated area.

Exercise 2a: Solution

```
class MouseEventLogger implements MouseObserver {
  readonly screen: number;

  constructor(screen: number) {
    this.screen = screen;
  }

  update_coordinate(coord: Coordinate): void {
    console.log(`Click: (${coord.x}, ${coord.y})`);
  }

  update_drag(start: Coordinate, end: Coordinate): void {
    console.log(`Drag: (${start.x}, ${start.y}) to (${end.x}, ${end.y})`);
  }
}
```

In JS/TS, `Hi ${name}` allows us to produce a string where the value in the variable name is inserted into the string. So if name = “Jaime”, then we would have the string “Hi Jaime”.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 2b: Solution

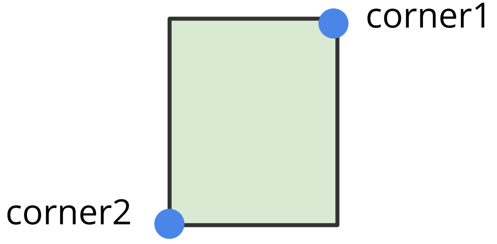
```
class MouseEventArea implements MouseObserver {
  readonly screen: number;
  private _min_vals: Coordinate;
  private _max_vals: Coordinate;

  constructor(screen: number) {
    this.screen = screen;
    this._min_vals = { x: Infinity, y: Infinity };
    this._max_vals = { x: -1, y: -1 };
  }
  // ...
}
```

To keep track of the rectangle containing all points clicked so far, we will track the minimum x and y values, as well as the maximum x and y values in two private attributes. We initialize the minimum values to Infinity and the maximum values to -1 so that they will always be updated after the first `update_coordinates` call.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 2b: Solution



```
class MouseEventArea implements MouseObserver {
  readonly screen: number;
  private _min_vals: Coordinate;
  private _max_vals: Coordinate;

  private calculate_area(corner1: Coordinate, corner2: Coordinate): number {
    return Math.abs((corner1.x - corner2.x) * (corner1.y - corner2.y));
  }
  // ...
}
```

We will define a private helper method to calculate the area of a rectangle given two of its opposite corners. The area is defined as the absolute difference between the x values times the absolute difference between the y values.
Taking the absolute value of the entire product is equivalent to taking the absolute value before multiplying.

Exercise 2b: Solution

```
class MouseEventArea implements MouseObserver {
  readonly screen: number;
  private _min_vals: Coordinate;
  private _max_vals: Coordinate;

  update_coordinate(coord: Coordinate): void {
    this._min_vals.x = Math.min(this._min_vals.x, coord.x);
    this._min_vals.y = Math.min(this._min_vals.y, coord.y);
    // ...
  }
  // ...
}
```

Given a new coordinate, we want to first update our min and max values accordingly. For the min values, we can do so using Math.min(), setting it to the minimum of the current min values and the given coordinate values. We can do something very similar for the max values.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 2b: Solution

```
class MouseEventArea implements MouseObserver {
  readonly screen: number;
  private _min_vals: Coordinate;
  private _max_vals: Coordinate;

  update_coordinate(coord: Coordinate): void {
    this._min_vals.x = Math.min(this._min_vals.x, coord.x);
    this._min_vals.y = Math.min(this._min_vals.y, coord.y);
    this._max_vals.x = Math.max(this._max_vals.x, coord.x);
    this._max_vals.y = Math.max(this._max_vals.y, coord.y);
    const area = this.calculate_area(this._min_vals, this._max_vals);
    console.log(`Coordinate area: ${area}`);
  }
  // ...
}
```

We then calculate the area with the min and max values, and print the area as specified.

Exercise 2b: Solution

```
class MouseEventArea implements MouseObserver {
  readonly screen: number;
  private _min_vals: Coordinate;
  private _max_vals: Coordinate;

  update_drag(start: Coordinate, end: Coordinate): void {
    const area = this.calculate_area(start, end);
    console.log(`Dragged area: ${area}`);
  }
}
```

Since we are just calculating the area of the rectangle formed by the start and end points of the current mouse drag, we don't need to deal with any min or max values. We can directly calculate the area using our helper method and print it out as specified.