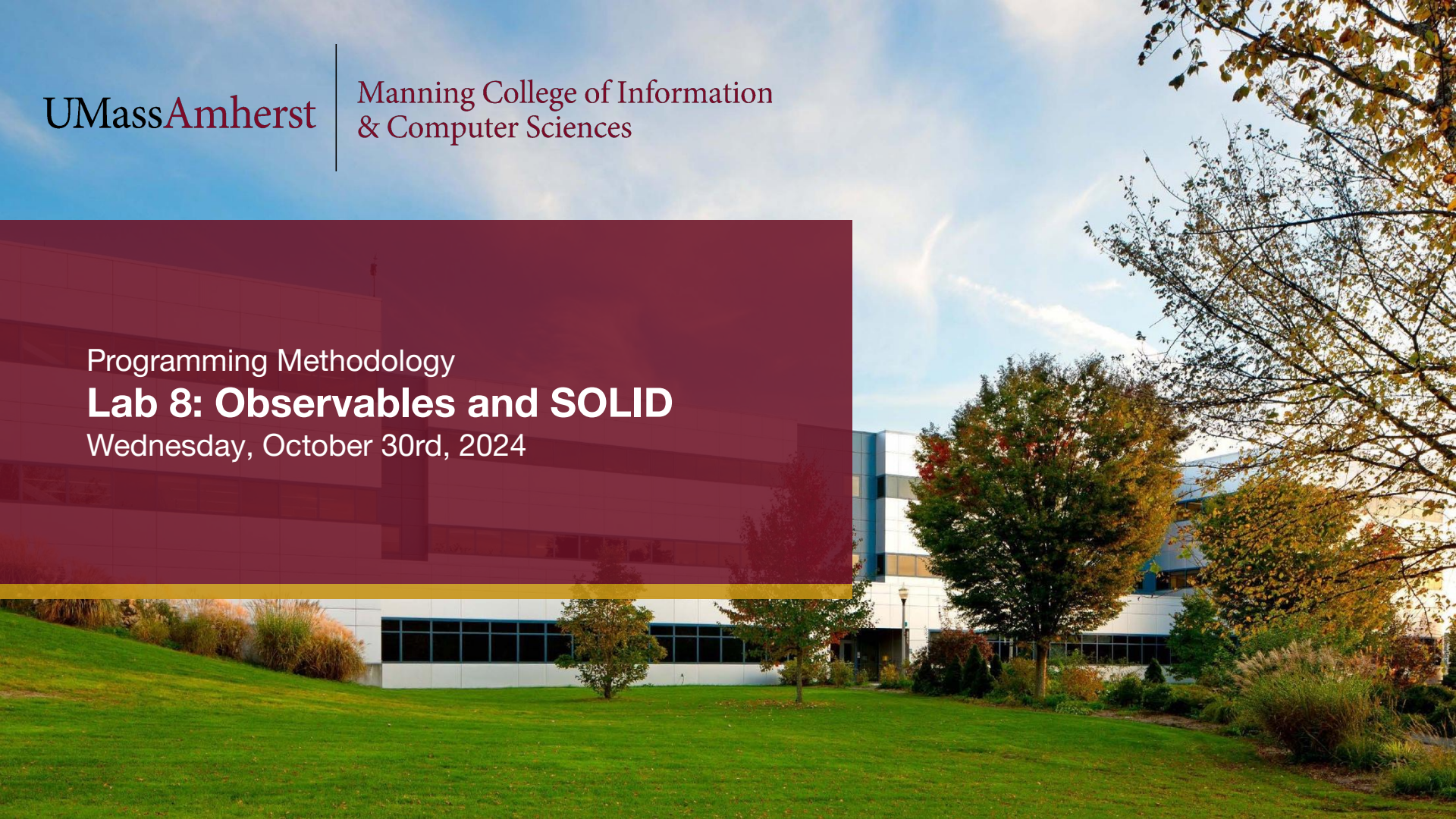


Programming Methodology

Lab 8: Observables and SOLID

Wednesday, October 30rd, 2024

Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
    - Please sit with your group from last week.
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- Download the starter code.
- Homework 6 is due tonight at 11:59pm
    - Come to office hours for help!
- The observables extra credit assignment will be released
    - Due Tuesday October 31st November 5 at midnight
- Complete the CATME Survey by next Friday November 3rd at midnight
- Midterm 2 is next week!
    - Start studying early.
    - Lab next week will be held as scheduled and attendance is required

Today's Goals

- Practice working with the observer pattern
- Practice working with streams

Observer Review

- What: A design pattern in which an observable subject automatically notifies dependent observers of any state changes
- Why: It's everywhere. E.g: GUI updates
- How: Reusable class

```
type Observer<T> = (x: T) => any;

class Observable<T> {
  private observers: Observer<T>[] = [];  // Maintain a list of observers

  subscribe(f: Observer<T>) {             // Add an observer to the list
    this.observers.push(f);
  }

  update(x: T) {                          // Notify each observer of update
    this.observers.forEach(f => f(x));
  }
}
```

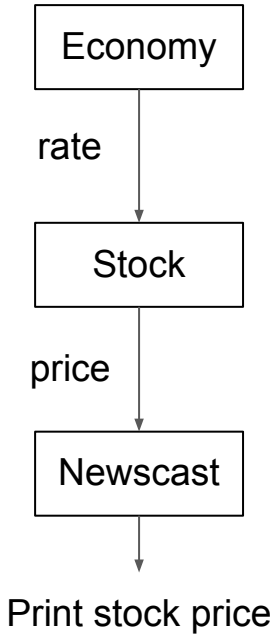
Exercise 1

- Model the stock market with 3 classes. Make sure to test!

```
// Should be "observable"
class Economy /* possibly extends something */ {
    updateRate(rate: number): void {} // Notify whoever cares about the economy
}

// Should observe Economy's rate, and be "observable"
class Stock /* possibly extends something */ {
    constructor(name: string, base: number) {}
    updatePrice(rate: number): void {} // Update price = base * rate
}

//  Should observe and report Stock's price
class Newscast {
    constructor() {}
    report(name: string, price: number): void {
        console.log(`Stock ${name} has price ${price}.`)
    }
}
```



‘Rest’ syntax

The **rest syntax** (`...`) in TypeScript (and JavaScript) allows a function to accept an indefinite number of arguments as an array. It’s useful when you don’t know in advance how many arguments will be passed to a function.

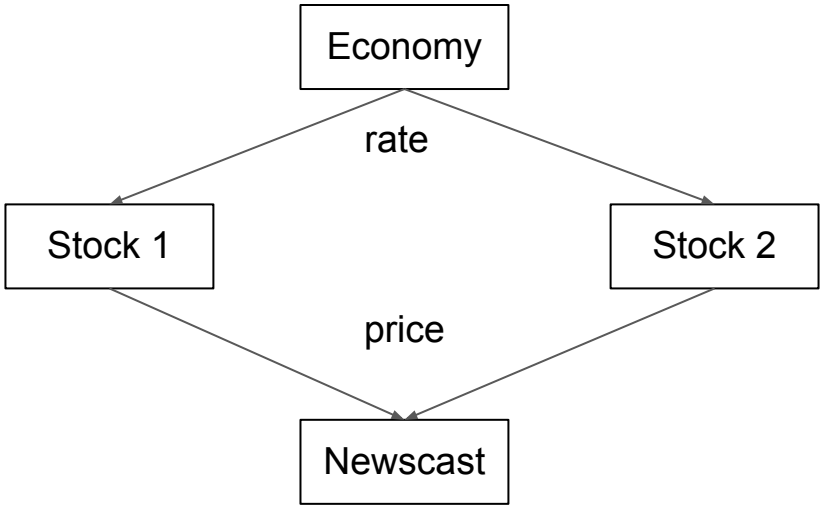
```
function multiplyAll(...numbers: number[]): number {
     return numbers.reduce((product, num) => product * num, 1);
}
```

Here's how you can use the `multiplyAll` function:

```
const result1 = multiplyAll(2, 3, 4);     // result1 will be 24
const result2 = multiplyAll(5, 10);       // result2 will be 50
const result3 = multiplyAll();            // result3 will be 1
```

Exercise 2

- Add a function `observe(...stocks: Stock[])` to `Newscast` so that it can observe any number of input stocks
- Make `Newscast` be an Observable that updates subscribers with the tuple [stockName, stockPrice] of type [string, number] whenever it reports



Exercise 3: Rectangle and Square

Does this class hierarchy satisfy the Liskov Substitution Principle?

```
interface Shape {
  area: () => number, perimeter: () => number
}
class Rectangle implements Shape {
  // use parameter properties shorthand
  constructor(private w: number, private h: number) {}
  area() { return this.w * this.h; }
  perimeter() { return 2 * (this.w + this.h); }
  getW() { return this.w; }
  getH() { return this.h; }
  setW(w: number) { this.w = w; return this; }
  setH(h: number) { this.h = h; return this; }
  symmetryAngles() { return [0, 90]; }
}
```

```
class Square extends Rectangle {
 constructor(len: number) { super(len, len); }
 setW(w: number) { super.setW(w);super.setH(w); }
 setH(h: number) { super.setH(h);super.setW(h); }
 symmetryAngles() { return [0, 45, 90, 135]; }
}
```

Give a code example where the expectations of the LSP are violated.
Restructure the hierarchy so the LSP holds.
You may introduce new classes, change method behavior (return new objects), etc.