

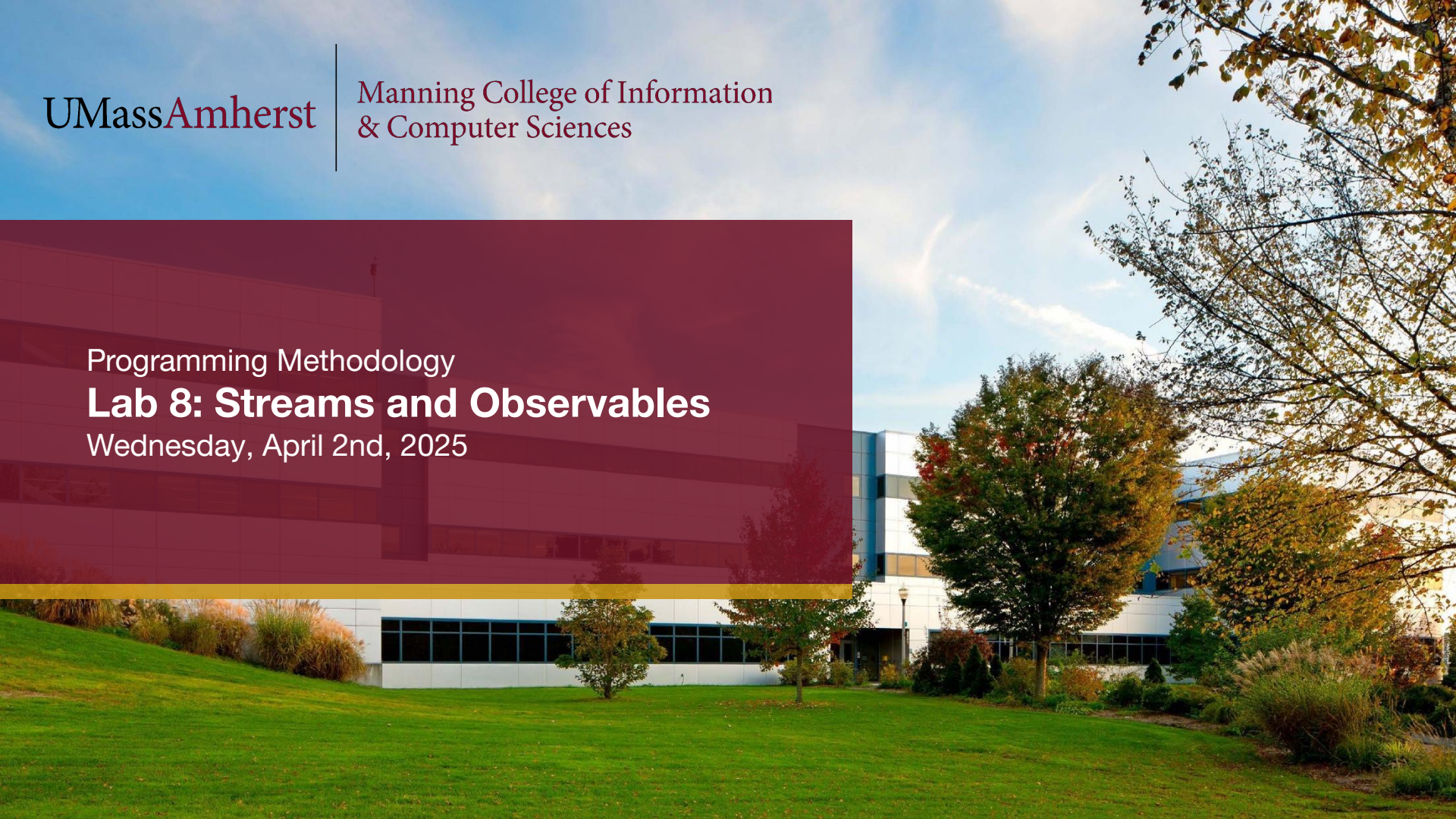
UMassAmherst

Manning College of Information
& Computer Sciences

Programming Methodology

Lab 8: Streams and Observables

Wednesday, April 2nd, 2025



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 (Streams) is due tonight at 11:59pm
 - Come to [office hours](#) for help!
- The Observables HW (extra credit) will be available soon
- Midterm 2 is next week!
 - Start studying early.

Today's Goals

- Practice working with Streams
- Practice working with Observables

Stream Review

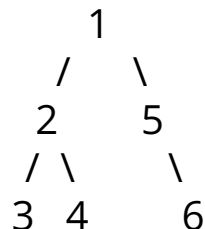
- What: A sequence of data made available over time
- Why: Useful abstraction for the paradigm where there's limited random data access and each data record can only be seen once*. E.g: Data reading, signal processing
- How: We implemented stream as a lazily constructed list with memoized tail

```
interface Stream<T> {  
  head: () => T;  
  tail: () => Stream<T>;  
  isEmpty: () => boolean;  
  toString: () => string;  
  map: <U>(f: (x: T) => U) => Stream<U>;  
  filter: (f: (x: T) => boolean) => Stream<T>;  
  reduce: <U>(f: (acc: U, e: T) => U, init: U) => Stream<U>; // This is new  
}  
  
reduce: (f, init) => snode(init, () => memoizedTail.get().reduce(f, f(init, head)))
```

Exercise 1: Preorder Traversal of Binary Tree

- Implement `preorderStream<T>(t: Tree<T> | undefined): Stream<T>`
- Input: A binary tree with the following type alias:
`type Tree<T> = { left?: Tree<T>; v: T; right?: Tree<T> };`
- Output: A stream of values that follow the prefix order (preorder) traversal of the tree.
- Example:

Input tree:



Output stream: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> empty()

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 2: Merge Observables

- Write a function

`merge(o1: Observable<string>, o2: Observable<string>): Observable<string>`

that returns a new `Observable<string>` which, whenever either `o1` or `o2` are updated, will be updated with the same value.